Start writing *tracking* programs right away using this friendly guide

[redacted]

Tracker

FOR DUMMIES®

4th Edition

GNU compiler and all code from the book on CD-ROM

A Reference for the Rest of Us!®

Claude A Pruneau

Author of MORE Tracking For Dummies®

STAR INTEGRATED TRACKER

Integrated Tracker Task Force

May 2003

1 Introduction

The integrated tracker task force (ITTF) was formed in November 2000, to develop a new tracking code for the STAR collaboration. The interest in a new tracker spurred from the realization that the existing tracker, written in FORTRAN, was increasingly difficult to maintain, and could not readily be adapted or modified to include tracking in detectors other than the STAR TPC. It also became obvious the tracker speed would render difficult the analysis of the very large datasets the STAR experiment was about to accumulate. Moreover, the ongoing commissioning of the SVT and FTPC was bound to compound the problem, increase the complexity of the code, and its running time. A new tracker was indeed needed: one that could deliver equivalent performance in terms of track reconstruction quality, but at much increased speed, and with better maintainability and flexibility. The new code shall be written with an object-oriented design, provide for easy upgrades, addition or substitution of components.

The functional, performance and implementation requirements of the tracker are presented in Chapter 2. These requirements form the envelope within which, loosely speaking, the new tracker was designed. The design began with the specification of the reconstruction algorithms and the elaboration of a conceptual object model. These are presented in Chapter 3 of this document. The organization of the code and the implementation of the conceptual model is described in Chapter 4. Chapter 5 presents the STI foundation classes used in the tracker. The hit model, its implementation, and the related tools needed to instantiate, store, filter and use hits are presented in Chapter 6. The track model, it representation and the related tools to instantiate, filter, and store tracks are discussed in Chapter 7. The detector model, its implementation, containers, builders, and related components are discussed in Chapter 8. A detail "how to" recipee describing how to include and deploy new detector systems in the tracker is presented in Chapter 9. The track finder and fitter classes are presented in Chapter 10. The tracker uses a toolkit to manage all key components required for tracking. This toolkit is described briefly in Chapter 11. A very brief summary of Kalman filter theory is presented in Chapter 12. Basics recipees of MCS and ELOSS calculations used in the tracker are summarized in Chapter 13. The charge of the Integrated Tracker Task Force is presented in 14.

1.1 About the Task Force

The STAR-computing leader formed the integrated tracker task force on Nov 13th, 2000 with the mission to design, test, evaluate, implement, and document an integrated tracker for STAR. The new tracker shall:

Provides highly efficient and minimum-contamination information on particles emitted into the STAR acceptance.
Incorporates all tracking detectors taking into account their detailed geometry, calibration, material location and thickness, as well as magnetic field effects.
Provides tools that allow extrapolating from one position on the track to any other position along the flight path of the particle with high accuracy (track extrapolation).

The full charge of the task force is listed in Appendix 3.

1.2 Task Force Members and Acknowledgements

The core task force initially included Ben Norman (Kent State), Mike Miller (Yale), and Claude Pruneau (Wayne State), who developed the core components of the system. Andrew Rose (Wayne State), and Manuel Calderon (BNL) joined the task force in 2002, and contributed to the evaluation of the performance of the tracker as well as the addition of various components targeted towards the integration of the tracker into the STAR main stream analysis. Maria Mora-Coral (Max-Planck-Institut fuer Physik) and Camelia Mironov joined the group in 2003. Maria worked on the development of the FTPC geometry, while Camelia work on the development of the Messenger package and the integration of the kink finder. Zbigniew Chajecki (Warsaw) most recently joined the group and contributed very efficiently to add and maintain diagnostics and performance evaluation tools.

Our gratitude goes to Jerome Lauret (BNL) who coordinated the integration of the tracker into STAR production code, and its review by STAR. It is fair to say this project could not have been completed without his strong support and dedication. We acknowledge the very helpful assistance of Helen Caines (Ohio State/Yale) and Rene Bellwied (Wayne State University) in the elaboration of the geometry model of the SVT. We also acknowledge the invaluable contributions of Karel Safarik and Yuri Belikov, both from CERN, who gave us the code they developed for the ALICE tracker, and from which we extracted many useful components that are now seamlessly integrated in the STAR ITTF tracker.

Finally, we are also indebted to the following STAR collaborators who have and are still contributing to the evaluation of the tracker: Dan Magestro (OSU), Fabrice Retiere (LBL), Jennifer Klay (LBL), Lee Barnby (KSU), Mark Heinz (Yale), Mercedes Lopez-Noriega (OSU), Richard Witt (Yale), Sergei Panitkin (BNL), and Zhangbu Xu (BNL).

1.3 Further Documentation

This document presents an overview of the requirements, conceptual design, and implementation of the STAR ITTF tracker. As such, it is not meant to present a full description of the tracker code implementation which is separately available online, on the STAR website at the following URL:

http://www.star.bnl.gov/webdatanfs/dox/html

2 Requirements

The requirements are formulated in terms of functional requirements, performance requirements, and implementation requirements. They are presented separately in the next three sections.

2.1 Functional requirements

The tracker must be designed to satisfy the following generic functional requirements:

- Design and use flexible and polymorphic interface to enable access to data from various detectors such as, in STAR, the TPC, SVT, SSD, and possibly other detectors such as the FTPC, and even the TOF, and EMC.
- Enable a certain degree of flexibility on the detector geometry in order to accommodate upgrades.
- Use a Kalman Filter/Fitter to account for track multiple scattering and energy losses.
- Allow for many-to-many point to track relationships
- Use full error matrices (covariance) in the handling of hits and tracks.
- Use a robust track model - unlikely to carry "nan".
- Allow for usage of different track models if needed by using an abstract track model interface.

2.2 Performance Requirements

The new tracker should:

- Enable good track reconstruction to optimize the reconstruction efficiency while minimizing ghost or false tracks.
- Handle hit errors properly so that fit chi-square are meaningful
- Be faster the existing Star tracker
- Make efficient use of memory to limit the size of objects - however emphasis is on "speed" and reconstruction quality.

2.3 Implementation Requirements

The code should be implemented and deployed according to the following requirements. The code shall be

- Written entirely in C++.
- Developed with an object oriented design.
- Multi-platform portable.
- Compatible with ROOT.
- Adhere to STAR code development standards.
- Documented as much as possible. Documentation to include class, method level specifications as well as usage examples.

- Archived using the STAR archival system.

3 Tracker Design

The tracker goals and design considerations are discussed in Section 3.1. The tracker algorithm is presented in Section 3.2 along with the conceptual detector and track model used in the design and implementation of this tracker.

3.1 Statement of the problem and design considerations

This tracker is meant to provide both track finding and fitting functionality. Hits from measured with various detector components must be associated to reconstruct particle trajectories, and fitted to determine the curvature, direction, and origin of the track. One must also, and more generally, determine the momentum and species identity of the particle.

The determination of the curvature is somewhat straightforward. A minor difficulty however arises when trying to reconstruct the momentum vector of the physical particle. From a physics standpoint, the momentum vector one seeks is the vector at the vertex of origin of the particle. The problem is that the point of origin can be any of the following:

- Main interaction vertex
- A spurious interaction vertex due to event pill-up
- Secondary vertex
- Decay vertex
- A scattering center

One is thus led to formulate a track reconstruction algorithm which makes no a priori assumption as to the origin of the particles: the assignment of the track to a particular vertex of origin must be done after the track parameters have been determined. Viewed as an object, the track thus consists of a collection of points acquired or found with the appropriate algorithm, a parameterization of the track based on a fit of the data points to a model or template, and a vertex of origin. Properties such as the momentum (modulus or vector), and the particle identity are then calculated afterwards on the basis of the track parameters, and the known position of the vertex of origin. Note that one can make assumptions about the vertex of origin, and include it in the fit for the determination of the track parameters after the fact, i.e. after it has been associated with the track.

One is then left with the core of the problem: finding the tracks, and fitting them to the chosen (and hopefully appropriate) track model to eventually deduce the particle final state. It thus appears natural to define a "tracker" entity whose purposes are:

- To find the tracks based on a store or bank of hits reconstructed within the relevant detectors.
- To fit the hits using a suitable track model.
- To enable association with a vertex of origin and optionally allow a refit of the data including the vertex of origin.
- To calculate the final state particle information.

The virtue of a Kalman Filter approach is to integrate in an efficient and compact way both the finding and fitting steps. One must however pay attention to “some” details...

In a detector such as Star, the track reconstruction in the TPC, SSD, and SVT, naturally proceeds from the outside to the inside. Track densities on outer layers of the TPC are smaller than on the inner layers, there is thus much less ambiguity in forming and following tracks. The Kalman approach enables to progressively use the points available to refine the knowledge of the track parameters, and extrapolate (follow) the tracks inward. The calculation of the track parameters and the extrapolation from layer to layer shall proceed according to the canonical Kalman filter algorithm described here. The finder however needs a sensible seed before it can proceed in finding tracks.

Given that the number of hits in the STAR detector can be rather large for a central Au+Au collision event, it is imperative one implements a hit data store which enables fast and efficient retrieval of the relevant points. The key word is relevance. The finder shall not have to iterate on all data points to find sensible candidates for the continuation of tracks. One should thus define a measured hit/point data store, which enable point retrieval based on a layered, coarse grain pixelization of the detector.

Additionally, given that as one follows the track into the inner TPC sectors, or the SSD and SVT, ambiguity may arise as to which point is best to add on a particular track. It may thus become appropriate to fan out the tracks and follow multiple leads concurrently.

The extension of tracks from the TPC to the SVT (or backward) across structures such as the inner field cage of the TPC raises the important issue of effects caused by multiple scattering and energy losses. Given that much of the particles detected by Star have low momenta, it is critical to include these effects properly in the propagation and fit of the tracks. We shall adopt much of the work done for the Alice detector by K. Safarik, and Y. Belikov.

The components, minimally needed, can be summarized as follows

- Hit entities that encapsulate the position, error, energy loss, or deposition of track in detector components.
- A hit container providing polymorphic hit data storage and ultra fast retrieval of hits based on a hierarchical, layered, coarse grain representation of the detector.
- Abstract track, which define the notion of track.
- Concrete Track entities implemented following the chosen track model to hold reference to hits associated with the track, and with accessor and modifiers properties to set and get the physical properties of the track.
- A track container providing polymorphic track storage and fast retrieval based on various sorting algorithms needed, for instance, in the analysis of track merging.
- Abstract Track Finder defining the notion of tracker.
- Concrete Track Finder implementing the Kalman track finder developed in the context of this project.
- Abstract track seed finder defining the notion of track seed finder.
- Concrete Track Finder implementing a local seed finder developed in the context of this project.

3.2 Tracking Algorithm

We have, in the past, explored a number of fitting algorithms for the reconstruction of tracks in a complex detector such as STAR. While global search methods based on Hough transforms, or track template may be deployed in very elegant, CPU efficient ways, and do well for the reconstruction of primary tracks, they typically do rather poorly in the reconstruction of secondary tracks – those produced from the decay of short lived particles, or from interaction within the detectors. Moreover, the application of template methods would require, for use with a detector such as STAR, a huge set of templates (even if the obvious cylindrical 12 sectors, two halves symmetry of the TPC is exploited) and would end up requiring a rather substantial memory allocation. Moreover, with such methods, as the track finding is completed, one still needs to perform a fit of the tracks that accounts for energy loss and multiple coulomb scattering effects. We have thus opted for a more conventional approach based on a Kalman filter.

We first describe the general track finding strategy in section 3.2.1. The track search and fit algorithm is summarized in Section 3.2.2. The track model, and the specifics of the Kalman finder/filter/fitter are presented in section 3.2.2.

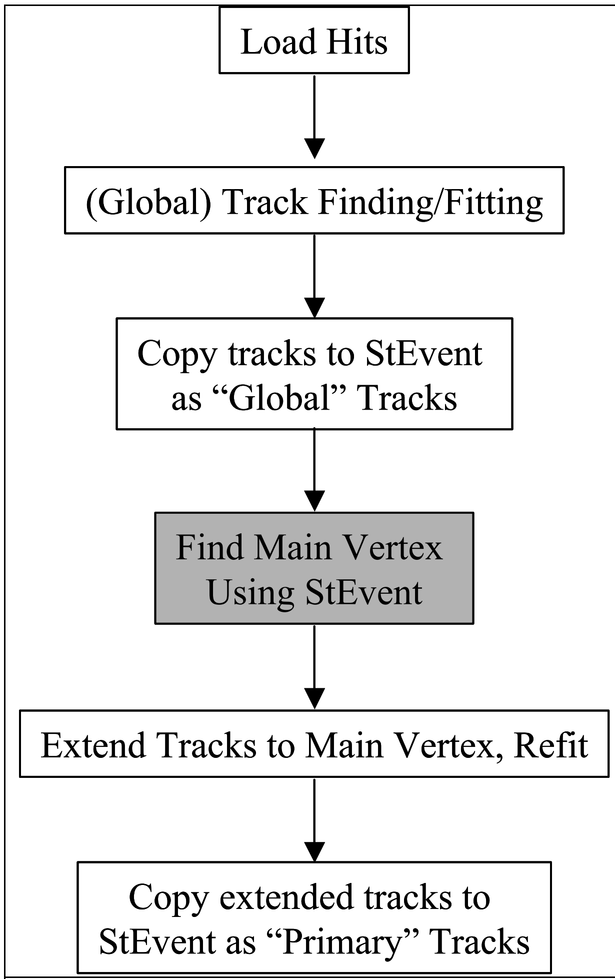

Load Hits
(Global) Track Finding/Fitting
Copy tracks to StEvent as "Global" Tracks
Find Main Vertex Using StEvent
Extend Tracks to Main Vertex, Refit
Copy extended tracks to StEvent as "Primary" Tracks

Figure 3-1 General Track Reconstruction Strategy. Sequence of tasks involved in the track reconstruction. Note that the main vertex is outside the scope of this project.

3.2.1 General track finding strategy

The methodology used for the track reconstruction is basically that of a “Kalman road finder”: given an existing segment of a track, use the knowledge provided by this segment, to predict and estimate where the next point on a track might be; once you got there, use the new point to update the knowledge of the track. Overall, the approach can thus be qualified as localized in space, or simply “local” by opposition to the global search techniques alluded to in the introduction of this section.

STAR uses the notions of global, primary, and secondary tracks. Primary tracks are those emanating directly from the main collision vertex whereas secondary tracks are produced by decay or interaction of primary tracks within the detector. The finite resolution of the track reconstruction, and kinematical focusing of decay products concur to render the distinction between many secondary and primary tracks rather difficult. STAR thus first analyze all tracks as if they were secondary tracks, and do not include the main collision vertex. One then search for the fraction of those that present a good match with the main collision vertex and can be labeled as primaries. The tracks obtained in the first pass are labeled “global tracks” and are fitted without a vertex. The primary tracks are extension of the global tracks including the vertex: their fit includes the vertex. Note that STAR maintains a double list of tracks consisting of global and primary tracks, where tracks that match the main vertex appear twice - once as global and once as primary. It is thus possible to save to disk, the track parameters with and without the primary vertex for further analysis of V0s and other decay topologies.

STAR uses an event model called StEvent. This event model also contains a track model called StTrack. As we started to develop this new tracker, we felt the STAR StTrack model did not provide the flexibility and efficiency need for this tracker, and we thus designed and implemented a new track model for within this tracker. Given that much of the existing STAR C++ code already use the StTrack model, we concluded it would be simpler to keep the existing track model for i/o purposes while conducting the track search with the StiTrack model. This implies that once StiTrack tracks have been found, they must be copied into the StEvent format.

The track search and event reconstruction algorithm, shown schematically in Figure 3-1, proceeds in basically five steps. The first step consists in the actual track search and is described in the following section. It produces “global tracks”, in the STAR jargon, i.e. tracks with no associated vertex. Those global tracks are then copied into the STAR event model StEvent/StTrack by a call to a filler helper class method. The main vertex finder is called next (with StEvent as argument) to find the vertex of the event. If a vertex is found, the Kalman vertex finder is called, once again, to attempt an extension of all found tracks to the main vertex. The event filler is then call once more to copy the newly found primary tracks, i.e. those tracks that were successfully extended to the main vertex. The track reconstruction is then completed.

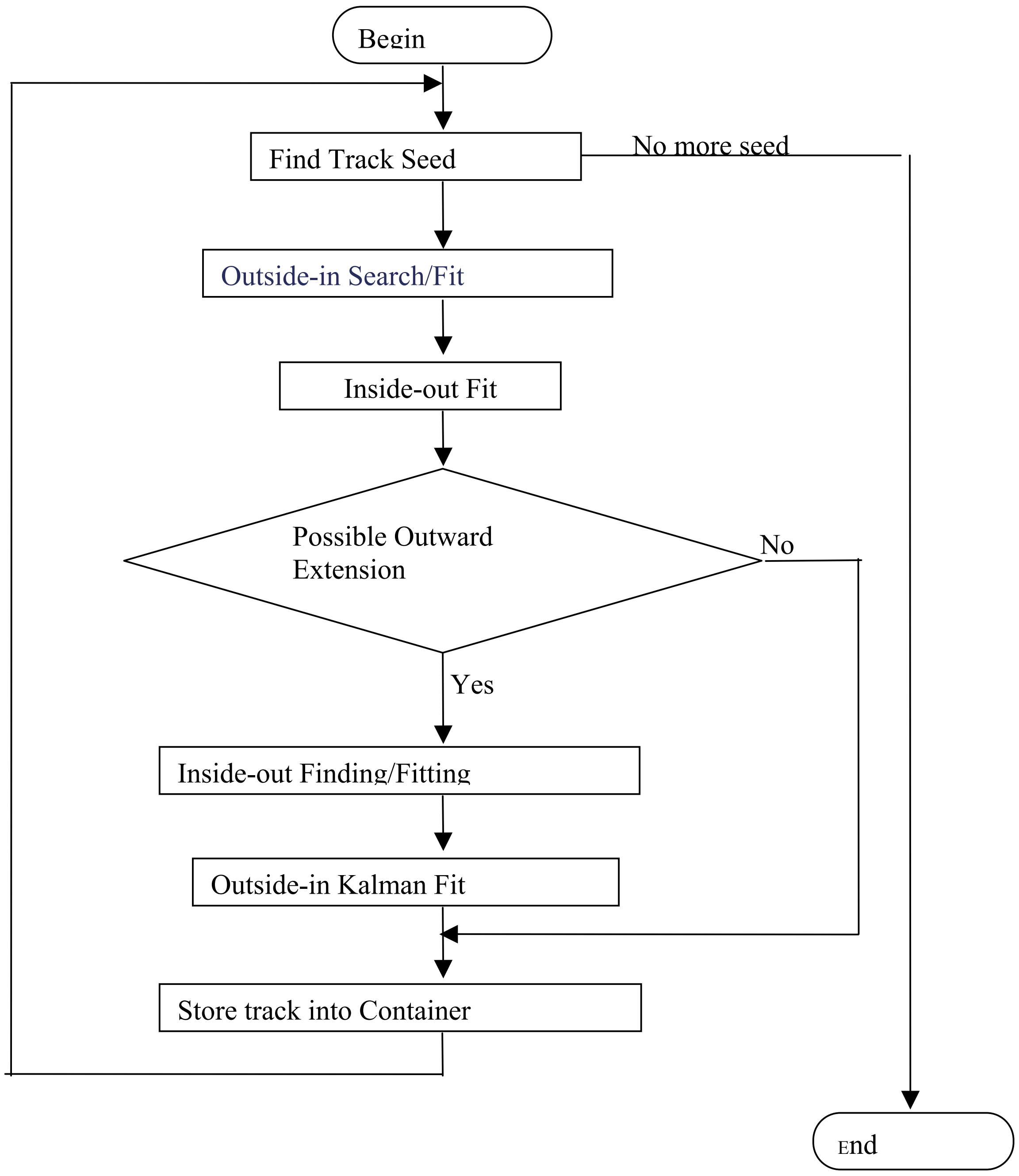


Figure 3-2 Track Search Algorithm

3.2.2 Track Search and Fitting Algorithm

The track search and reconstruction algorithm is illustrated schematically in Figure 3-2. The search uses a Kalman road finder, and proceeds, in a loop, sequentially, track by track until no more tracks are found. No correlations between tracks are considered although hits may initially belong to more than one track.

The search for each track is initiated with a call to a Track Seed Finder. The search stops when the seed finder returns no seed. Track seeds are short track stubs consisting of a sequence of a few hits. As such, they carry just enough information to enable a very rough estimate of the track position, direction, and curvature. This rough estimate will be used the Kalman finder to begin the extension and search of the track through the detector. Seeds returned by the seed finder are not confined to any specific region of the detector. Given however, it is easier to find reliable track patterns in a low track density environment, the search for seeds proceeds, in the STAR detector, from the outside in. So, typically the seeds returned are towards the periphery of the detector. The Kalman search that follows thus first proceeds inward. The algorithm of the seed finder is discussed in more details in Section 3.2.3.

The Kalman-search proceeds through the virtual layers of the detector, step by step. It is considered complete when the search reaches the inner most volume, or when a prescribed minimum number of active detector layers have been crossed without finding matching hits. The mathematical details of the Kalman search and fit are described in Section 3.2.7. The Kalman finder uses the direction and curvature of the existing track stub to estimate (extrapolate) the position of the next track hit on the next available layer.

Matching hits are then sought on that layer within a radius of confidence determined by the error parameters of the track. If no matching hit is found, the given layer is skipped. If one or more matching hit candidates, one calculates the increment of track chi-square caused by the addition of the candidate hits. Candidates are deemed acceptable if the chi-square increment is smaller than a prescribed (user settable) maximum. If more than one candidate hit satisfy the chi2 requirement, one selects and add to the track the hit with the lowest incremental chi-square value. Once a hit is added, the track parameters (i.e. curvature, direction, etc) are updated using the Kalman track model discussed in Section 3.2.6. As the track-search proceeds inward and eventually reaches the inner most detector volume, the track parameters are progressively refined and précised. The Kalman parameters (including the chi-square) of the track at the last hit are the best estimator of the track.

Given that the track search initially proceeds on the basis of a seed that may not lie at the very edge of the detector, it is possible that the track found after the inward pass might be incomplete. One thus test whether the outmost point on the track is sufficiently far from the edge of the detector that points might potentially be added to the track where a search conducted in that part of the detector. The search is considered complete if a number of points smaller than a prescribed minimum could be added. It otherwise continues. The continuation of the track outward proceeds similarly to the inward pass. Successive virtual layers are search step by step for additional hits, and the track parameters are updated at each step. Note however that in order to initiate the outward pass, an outward refit of the track is first performed in order to update the track parameters of the outer most node of the track. The fit is performed with the same machinery (methods) than those used by the finder. The only difference lies in the fact that the hits are already found, so one only needs to update the track parameters. The outward search proceeds until the edge of the detector or until too many layers have been crossed without association of hits on to the track. The same threshold is used here as for the inward pass.

If an outward pass is performed, and once completed, the track parameters of the inner track nodes can be considered under constrained since not all hits on the track were used to calculate the track parameters for those nodes. An inward track refit is thus accomplished.

If an outward pass is not performed, the track parameters of the outer nodes can also be considered under constrained. An outward final fit is thus conducted. This fit is deemed necessary to provide best track parameter knowledge on the outset of the track, which may then be used by user analyses for extension of the tracks to non-tracking detectors such as, in STAR, the CTB, the TOF, or the EMC.

3.2.3 Track Seed Finder

The seed finder is responsible for finding some portion of a track given a collection of hits. The track segment thus found is then passed to the actual finder, which extends the tracks through the entire detector volume. The role of the seed finder is critical: it must enable the recognition of primary, secondary, low, high momentum tracks without tracks without biases.

Many track pattern recognition algorithms exist. These algorithms can be separated roughly into global and local algorithms. Global algorithms (sugh as Hough transforms, neural nets, etc) tend to be O(N^2) algorithms, where N is the number of hits. Local algorithms tend to be much better, O(N). Further, a local algorithm lends itself to Kalman filtering. We have thus developed a local seed finder. The charge of ITTF requires enabling easy upgrades or test of other algorithms. We also considered that optimization of the tracker might require multiple techniques be used for seed finding. We thus adopted a design where multiple seed finders could be used sequentially. In essence, all seed finders to be used shall derive from a base class defines the notion of seed finding. A composite seed finder class, which consists of a container of send finders, is then use to broker the actual finders in doing the seed finding work.

We have implemented a local seed finding approach that goes by the name of "road finder" or "follow your nose" tracker. Essentially, it identifies two points that are close in position space. From these two points it extrapolates to another layer using a straight-line trajectory. At the next layer it adds another hit and then moves on. The process continues until the seed is either a user specified minimum length (number of hits) or aborted. Fast circle (in the plane transverse to the field) and linear (path length vs. z plane) fits are used to estimate the parameters of the track, and initialize the Kalman state passed on to the Kalman track finder. The seed finding process is iterated until all hits have been visited.

3.2.4 Conceptual Detector Model

The Kalman filter tracker developed in this work implicitly requires the knowledge of the location, size, orientation, and material composition, of the detector components, and other material structures present in, or near, the fiducial volume where charged particle trajectories are measured. It is necessary to account for the finite density and thickness of the materials traversed by particles to a local basis: one needs to know, at each track step, what volume are crosses, to determine an estimate of the Kalman process noise (MCS) and energy loss. Because STAR is a detector in somewhat constant evolution, and because we felt it would be interesting to consider the applicability of this tracker to other experiments, we decided it would be appropriate to generate an abstract model to describe the detector geometry rather than hard coding the necessary material information.

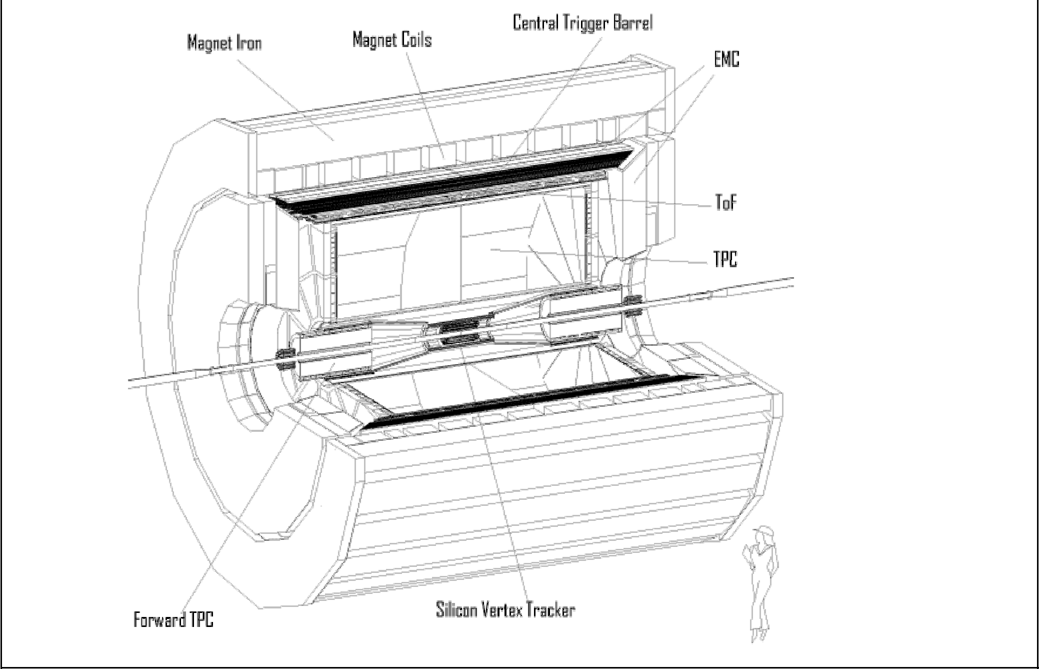


Figure 3-3 Cut away view of the STAR detector

The STAR detector, illustrated schematically in Figure 3-3, features a rather simple cylindrical geometry. The central detectors, which is the primary focus of the tracker developed by the ITTF group, features a discrete azimuthal symmetry, and a rather simple radial ordering of detector components. Such a simple geometrical structure suggests one can account for the presence of scattering materials within the detector while maintaining a rather simplistic, and coarse model of the detector. Essentially, our aim is to avoid using the GEANT geometry model, which although well proven in the business of detector simulation, implies a rather high CPU cost for the propagation of tracks across the detector volumes. We thus initially formulated a simplified geometry model whereby the detector components are organized in radially concentric virtual layers. In this model, the virtual layers may themselves be segmented azimuthally and longitudinally. This model readily accommodates the STAR central detectors and may also be adapted to model the forward detectors (such as the FTPC). For modeling of the forward detectors, one substitutes the z-axis to the radius and model the forward tracking planar detector components to be perpendicular to the beam direction.

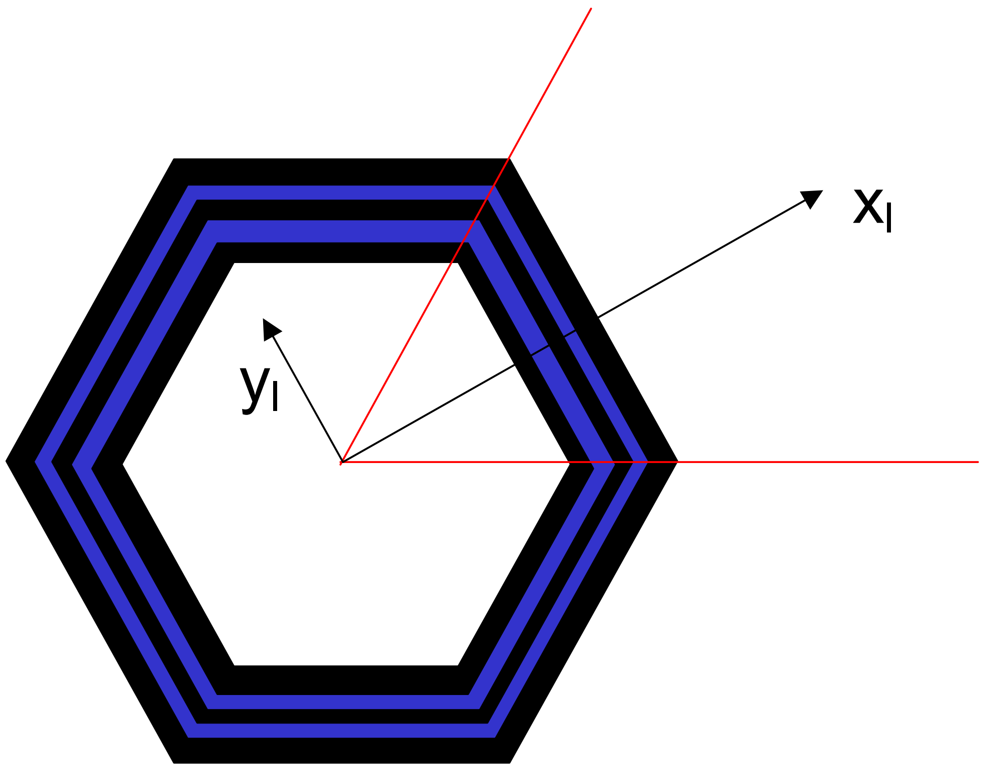


Figure 3-4 Schematic representation of the virtual layers and layer segmentation of a fictitious six-sector detector.

As schematically shown in Figure 3-4, the geometry model consists of successive layers locally transverse to a depth axis, which can be either radial, as in the case of the SVT, SSD, and TPC, or longitudinal as in the case of the STAR FTPC. The layers can be segmented in one or two dimensions. In the STAR TPC, each layer is segmented in 12 equal planar partitions azimuthally and no longitudinal segmentation is used.

The geometry model is based on a hierarchically organized system of elementary modules representing detector elements or inactive components with the detector fiducial volume. The hierarchical organization is achieved by inserting the modules into a smart container. The container maintains references to the elementary modules and enables their organization (or sorting) in successive layers, with possible segmentations within the layers. The detector module model involves a Boolean flag specifying whether the instance represent an active detector volume, whether the volume is a continuous medium, such as the gas within the TPC, or a discrete scatter, such as the SVT ladders, or the TPC field cage. The volume elements are characterized by a shape, a placement (position and orientation), the volume may contain a gas or fluid, and a solid material. Volumes are also given a name.

We have not attempted the same level of generality accomplished in the GEANT package, and have restricted the definition of volume shapes to rectangular and circular/cylindrical objects. These are sufficient to represents the volumes and constructs within the STAR detector.

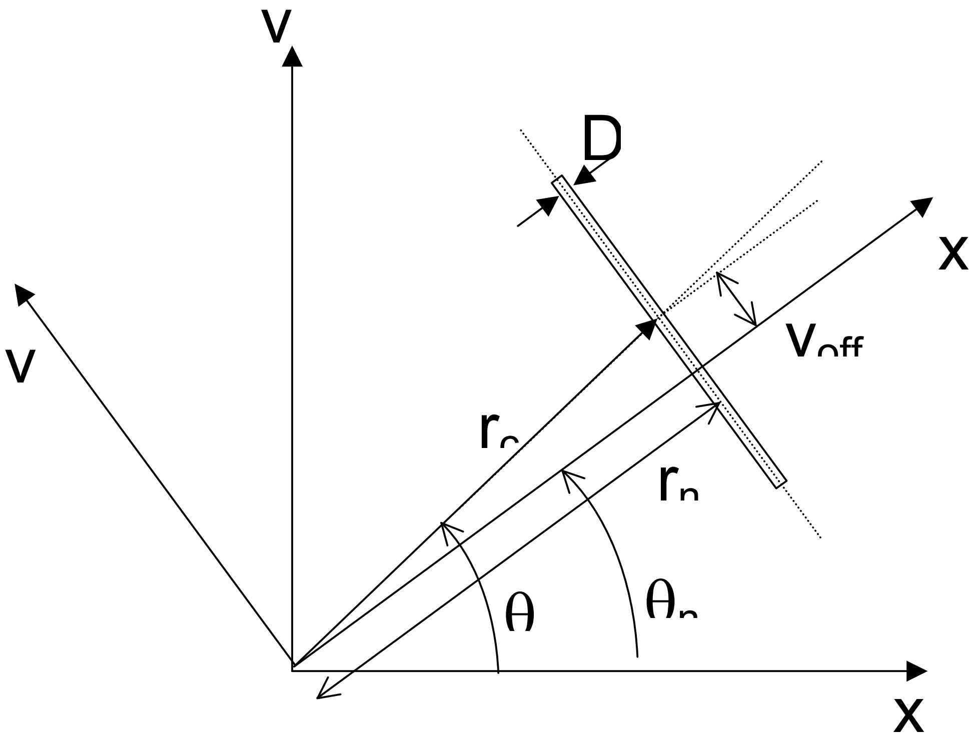


Figure 3-5 Detector Placement, The label "g" and "l" stand for global and local coordinates respectively. The placement of a detector element relative to the origin is formulated in terms of normal radius, r_n, and angle, θ_n , or in terms of the center position r_c, and θ_c.

Rather than having many small successive volumes to represent the various components of rather complex such as the SVT, we have found it sufficient to define rectangular volumes (e.g. ladder) that involve a large, mostly empty volume, and actual solid components. An SVT component is for instance represented as consisting of a rectangular gas volume within which lies a finitely thick silicon wafer. The navigation within the detector is thus greatly accelerated because one essentially jumps from one critical component to another while simplifying the propagation of the tracks through the gas and other continuous volumes. The calculation of the error matrices, in particular, is then performed, in a given step, by accounting for both the continuous nature of the material within a volume, and the presence, if any, of a discrete scatterer.

The placement of the detector is achieved with the representation illustrated in Figure 3-5. The "center" of a planar detector is its center of gravity (the midpoint in local x, y, and z). For curved cylindrical or conical sections) detectors, the "center" is the midpoint in z, opening angle, and radial thickness. The "normal" coordinates give the magnitude and azimuthal angle of a normal vector from the global origin to the plane of the detector. The plane of a planar detector is simply the one in which it lies. For a curved detector, the plane is the one that contains the tangent to the curved section at its center and is normal to the transverse projection of the radial vector from the origin to its center. Note that these definitions all assume that the plane of the detector is parallel to global z. The third "normal" coordinate gives the location of the detector center along the detector plane in the azimuthal direction (i.e., local y). This representation is best for the Kalman local track model. The "center" coordinates are a little more natural and are best used for rendering and radial ordering. Here, the magnitude and azimuthal angle of a vector from the global origin to the center of the detector are given, as well as an orientation angle. The orientation angle is the angle from the vector above to the detector plane's outward normal. It is 0 for detectors that have xOffset==0 when setting the values, one must set all 3 for a representation at once. The layerRadius is independent and is used for ordering detectors in R.

The information required for materials include their density, and their radiation length thickness. It also is convenient to label them with a name. Other data used in GEANT are not necessary since one does not actually consider interactions of the tracks in the media other than multiple Coulomb scattering and energy loss.

3.2.5 Conceptual Hit Model and local reference frame

Hits are measured in the local detector reference frame defined in the previous section. We assume the hit information determined by the detectors include a 3-D position, a full error matrix (or at least an estimate of the error matrix), the deposited energy, and a reference to the detector where the hit was produced. A translation to global coordinates must also be possible.

3.2.6 Conceptual Track Model

The track search is conducted within a local coordinate system, i.e. a coordinate system attached (local) to the detector components traversed by the track. For operation within STAR, we assume the magnetic field is perfectly constant and axial. This assumption could however be relaxed by the simple addition of a field map were the field found to have significant variation over the volume of the detector fiducial region. Charge particles are thus assumed to travel along helicoidal trajectories with a radius (and all other parameters) that may vary due to interactions and energy loss.

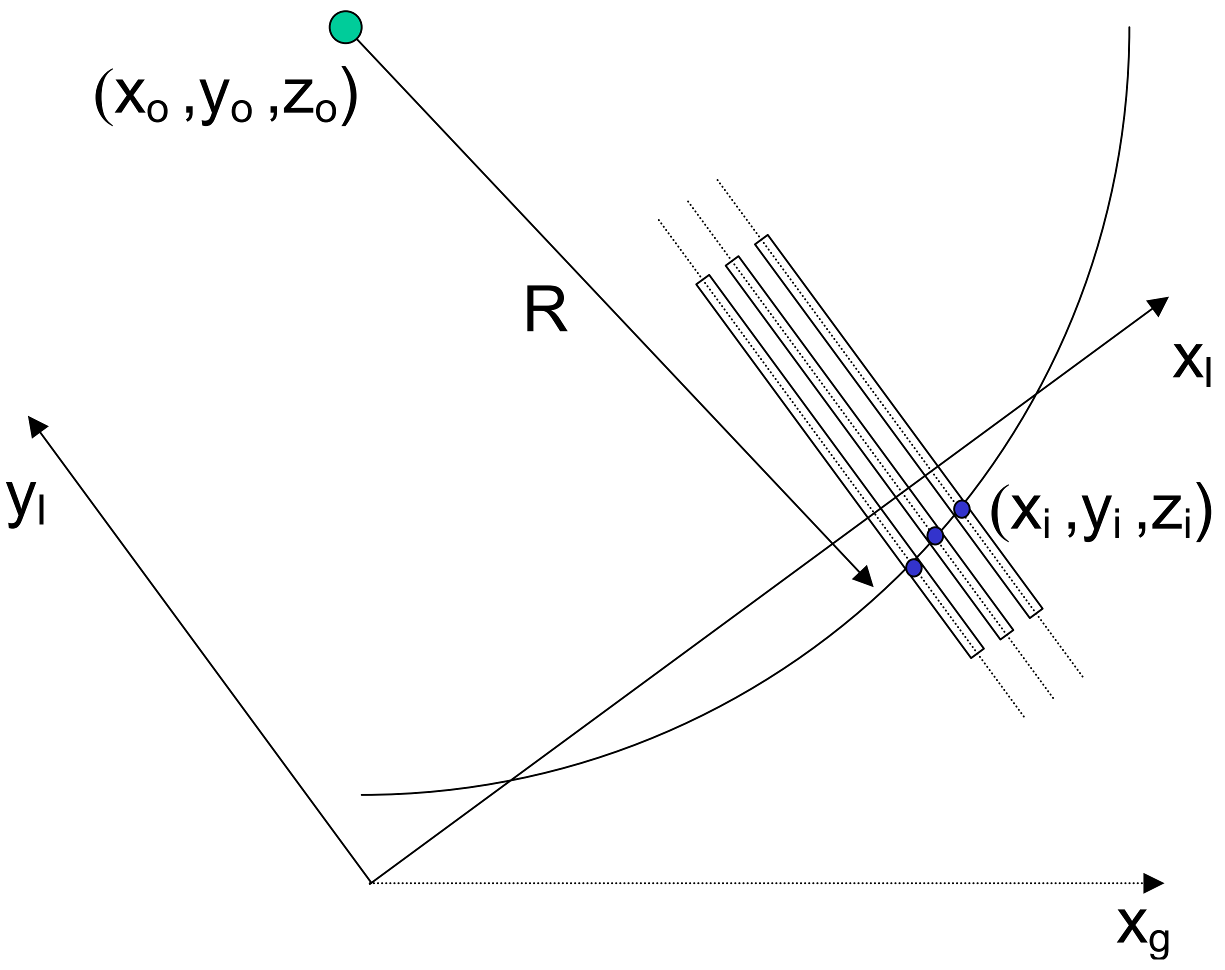


Figure 3-6 Local Track Model

The track model and its parameters are illustrated in Figure 3-6. The x-axis is used as the independent variable to measure the position of the hits from the origin of the reference frame. The detector components are assumed to be in the "yz" plane as illustrated. Parenthetically, we point out that having hits measured in a plane perpendicular to the independent variable simplifies and greatly speed up the calculation of residues.

At any given point along the track, the helix can be characterized, in the local reference frame, by stating the position of the center of the helix (x_0, y0) in the plane transverse to the B field, a reference position z_0, the curvature of the track, and the pitch angle λ. Alternatively, given that y_0, and z_0, are not directly measured but must be inferred from measurements, using

$$y_0 = y(x) + \tfrac{1}{C}\left(1-(Cx-\eta)^2\right)^{1/2}$$
$$z_0 = z(x) + \tfrac{\tan\lambda}{C}\sin^{-1}\left(Cx-\eta\right)$$

It becomes more advantageous to characterize the state of a track at given value of x, using x_0 (which must also be inferred), y(x), measured directly as the position of the hit along the "pad plane" direction, and z(x), measured directly along the **B** field direction, the curvature of the track, C, and the pitch angle λ. For computational purposes, it is actually more convenient to replace "x_0" by a related quantity, η, defined as η=C x_0. The state vector of a track at any given position "x" is thus represented as:

$$\begin{pmatrix} y(x) = y_0 - \tfrac{1}{C}\left(1-(Cx-\eta)^2\right)^{1/2} \\ z(x) = z_0 - \tfrac{\tan\lambda}{C}\sin^{-1}\left(Cx-\eta\right) \\ \eta = Cx_0 \\ C \\ \tan\lambda \end{pmatrix}$$

3.2.7 Kalman Search and Fit Algorithm

The Kalman track search and fit algorithm is schematically illustrated in Figure 3-7. It uses the track model presented in section 3.2.6. The search begins with a given track seed and proceeds iteratively with the addition of successive points. The seed may consist of an arbitrary number of hits as long as it is possible to calculate, from those hits, a reasonable estimate of the track parameters or Kalman state vector. Essentially, the search proceeds by first extrapolating the existing track to the next volume, search for possible matching hits in the vicinity of the extrapolation, calculate the incremental chi-square associated with the addition of points, select and use the best hits (lowest chi-square) to update the track Kalman state vector at the new hit position, and repeat iteratively.

The extrapolation of a track to a new location is done in two steps, and is driven primarily by the detector geometry. Given a hit (and its parent detector component), one must first find which detector element is susceptible of holding the next hit on the track. Currently, one assumes the track is either moving inward or outward, so one scans all detector/volume elements below (or above) that could host the next hit. Note that we plan to modify the volume scan as to permit successive hits on a track to be within the same virtual layer. The scan is done using a fast extrapolation to the center (in x) of the candidate volumes. One then selects for further inspection those volumes in which the extrapolation fall within or near the perimeter (in the yz plane) of the active region of the detector and volume. Note that the extrapolation falls well within the volume of a non-active volume (e.g. the TPC field cage), the scan for hits is skipped, and a no-hit node will be added to the track. The track current position will next be updated to reflect the new position, but the curvature, □, and tan□ are not changed given no new information is available. The Kalman track error matrix will be updated to account for the track propagation through the current volume. Once the next detector is found, one uses the current track state to predict the track position (x,y(x),z(x)) in the measuring plane of that detector. The prediction is based on the following expressions:

$$y_{i+1} = y_i + f(x)$$
$$z_{i+1} = z_i + f(x)$$
$$\eta_{i+1} = \eta_i$$
$$C_{i+1} = C_i$$
$$\tan\lambda_{i+1} = \tan\lambda_i$$

The predicted position is then submitted to the hit container to query for hits in that detector that may lie with a calculated search radius of the position. The calculated search radius is a function of the estimated track error, and scaling factor preset by the user. Note that lower and upper bounds are imposed on the calculated search radius in order to avoid pathological behaviors.

The hit container returns a list of hit candidates. If the list is empty, one treats the volume as a non-active volume; a no-hit node is added to the track, the track is propagated to the new position, and the error matrix update to account for scattering within this detector. If the list contains one or more hits, one iterates through all hits to calculate the incremental chi-square caused by the addition of the given points to the track. The hit with lowest incremental chi-square is selected as the best candidate. One then verifies the incremental chi-square is smaller than a preset maximum. The preset maximum is determined at run time, from user or external input. If the best candidate does not pass the maximum chi-square criteria, the detector volume is treated as inactive, and a no-hit node is added to the track. If the best hit candidate satisfies the criteria, the hit is inserted in a track node, and the track node added to the track. One then proceeds to update the track Kalman state at that node. The state update is calculated using the following expressions:

$$\begin{aligned} y_{i+1} &= y_i + f(x) \\ z_{i+1} &= z_i + f(x) \\ \eta_{i+1} &= \eta_i \\ C_{i+1} &= C_i \\ \tan\lambda_{i+1} &= \tan\lambda_i \end{aligned} \qquad \text{Equation 3-1}$$

The track error matrix is updated using the following equations:

Missing text.

Once this is completed, the tracking step is complete, and the process repeats with a scan for the next detector volume traversed by this track. The search stops when no further detectors are found.

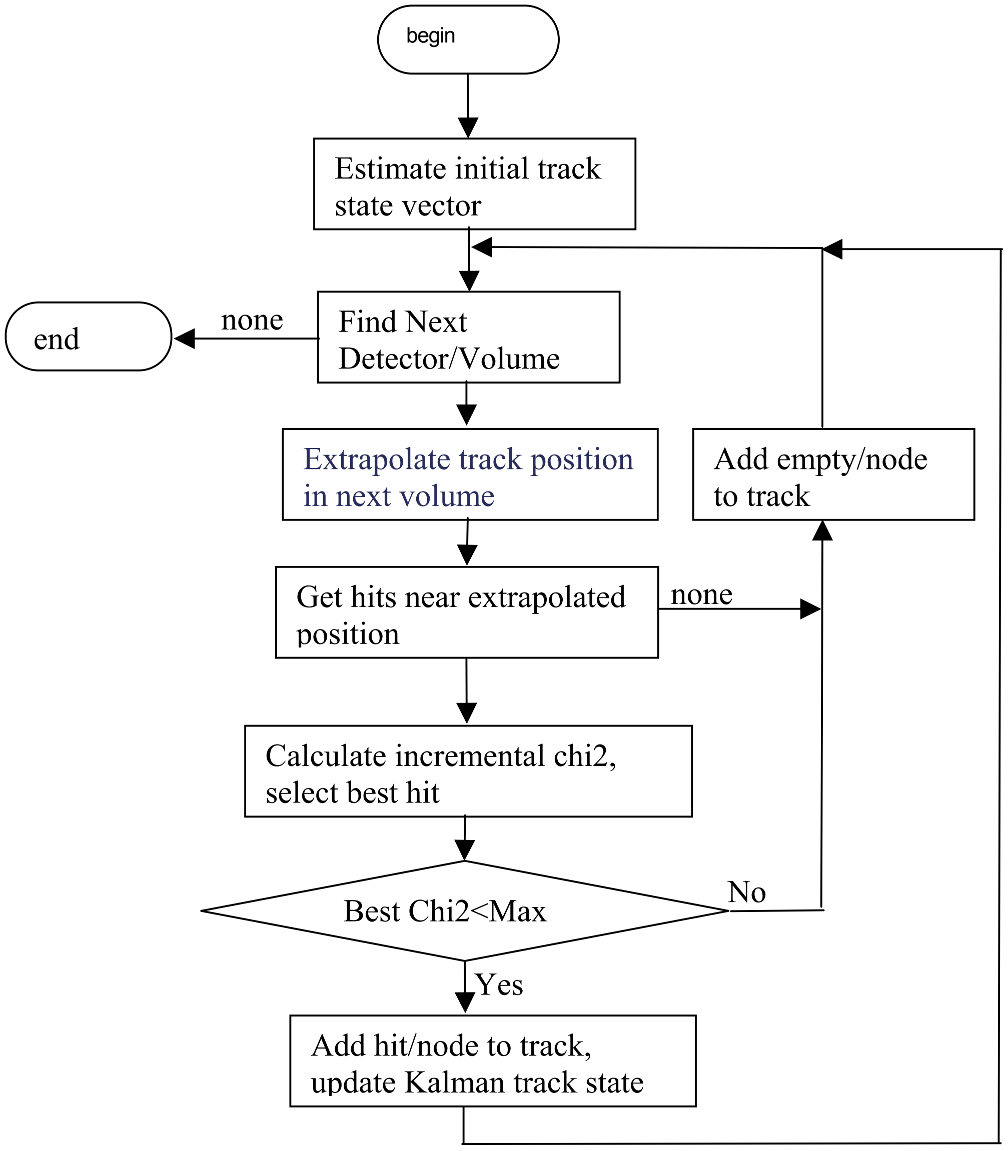

begin
Estimate initial track state vector
end
none
Find Next Detector/Volume
Extrapolate track position in next volume
Add empty/node to track
Get hits near extrapolated position
none
Calculate incremental chi2, select best hit
Best Chi2<Max
No
Yes
Add hit/node to track, update Kalman track state

Figure 3-7 Kalman Search and Fit Algorithm

4 Implementation Basics

4.1 Introduction

We present the general organization (and the motivations for this organization) of the code and packages layout in section 4.2. Given the need and requirements for an object-oriented design, we designed and developed the code using an object model. The organization, and general structure of the object model are presented in Section 4.3. Specific key components of the object model are presented in Section 4.4. We note finally that we have made a conscious effort to provide documentation within the code although critics will probably (always) consider it insufficient… The in-code documentation is accessible through the STAR website DOXYGEN generated documentation system.

4.2 Packages Layout and General Code Organization

The ITTF code is partitioned in four modules or packages called Sti, StiMaker, StiGui, and StiEvaluator. Sti constitutes the core package and contains the basic components (C++ classes) as well as the work classes (e.g. track finder, fitter, track seed finder, etc.) Graphical user interface (GUI) components used for the event display and settings of the code parameters in interactive mode are part of the StiGui package. Star and Root specific components are found in the StiMaker package. The StiEvaluator package provides performance evaluation tools convenient for the development and tuning of the code but not essential for its operation in production analysis. For operation within Star, the ITTF packages are used in concert with STAR specific (predating ITTF) packages such as StEvent, StMcEvent, StMiniMcEvent, StMiniMcMaker as well as the Star Class Library.

4.2.1 Sti Package

By design, the classes included in the core package "**Sti**" are meant to be independent of STAR experimental specificities such its geometry, and other software components used in event reconstruction. In practice, given ITTF software must be integrated with the STAR mainstream software, and to be able to communicate with the existing STAR software components, we have taken some license with this rule, and the STI classes do carry some dependencies on other STAR software libraries such as the StarClass Library for use of StThreeVector, StFourVector and other similar components. Also, to simplify the development of the code, at least at this stage, some of the Sti classes do carry some "hard coded" knowledge of the Star detector and software. Note however that as a part of our maintenance of this project, we shall endeavor to eliminate or abstract out those explicit dependencies and rely on derived classes implementing a "Builder" pattern to realize the Star specificities in this tracker.

The Sti package is written in standard C++, and has by design zero dependencies on non-Star packages. ROOT classes, for instance, are explicitly excluded to avoid, or at the very least minimize maintenance problems and dependencies on ROOT. ROOT classes and components are however used in the Star specific packages StiMaker and StiGui. Containers classes are used proficiently in this tracker, and one could have elected to use ROOT containers. We felt however that the adoption of ROOT containers would entail dependencies on the entire ROOT environment through the TObject base class used in all ROOT classes. We also felt it would be wiser to rely on industry standards for such containers and thus make ample use of the Standard Template Libraries (STL) in addition to the basic core C++ libraries distributed with most ANSI compliant compilers. We stress that the use of templated classes and of STL classes in particular enable a level of abstraction otherwise difficult to achieve in ROOT.

The Sti packages carries limited dependencies on Star software to facilitate the interface with other Star software components. Explicit references are made, within Sti classes, to classes such as StEvent, StMcEvent, StTrack, and StHit. Although it would be necessary to readily abstract out theses dependencies to a Star specific package to render the ITTF tracker truly "universal", we felt the urge and necessity of developing a new tracker for Star in a timely fashion outweighed the need for a universal tracker. We have thus included the Star Library classes and some other specific Star constructs to leak in our design.

4.2.2 StiMaker Package

As its name suggests, the StiMaker package provides a "maker" to operate the ITTF tracker within the Star software environment. It also includes a number of auxiliary classes such Star specific operations, and for event display. The StiMaker uses a class called StiDefaultToolkit to instantiate all relevant components at run-time based on a restricted number of control parameters. Such control parameters include flags to request operation of the tracker in Event Display mode, and in evaluation mode.

4.2.3 StiGui Package

This package comprises classes needed for the deployment and operation of the event display. The classes defined provide the means to display detector components, hits, and tracks.

4.2.4 StiEvaluator Package

This package was developed during the initial stages of the ITTF project development to provide a performance analysis of the track reconstruction. It has now been essentially replaced by the use of the STAR StAssociationMaker framework developed outside of the scope of this project for the determination of reconstructed track quality and efficiency. It is still used to provide a coarse evaluation of the tracker performance but it should be considered deprecated and it will eventually no longer be maintained.

4.3 Object-Oriented Model

The tracker code was created with an object-oriented design and is based on the algorithm and conceptual object model presented in Section. The developed C++ object model involves a large number of classes that can be, roughly speaking, organized in a tier system as illustrated in Figure 4-1. The model includes a number of abstract classes (some of which are pure abstract but not all are) used to define the interface to the object and constructs used in this project. However, we did not strive to define a pure abstract class for all constructs used in this tracker. This point is elaborated in Section4.3.1. Elementary and utility classes, part of the first tier, are described in Section . Data and geometry entities used by the tracker are briefly described in Section 4.3.2. Simple functors, helper and convenience classes developed for the purpose of simple calculations, and filters, constitute the second tier family of classes and are described in Section X. The code makes ample use of simple and containers and some “smart” containers.

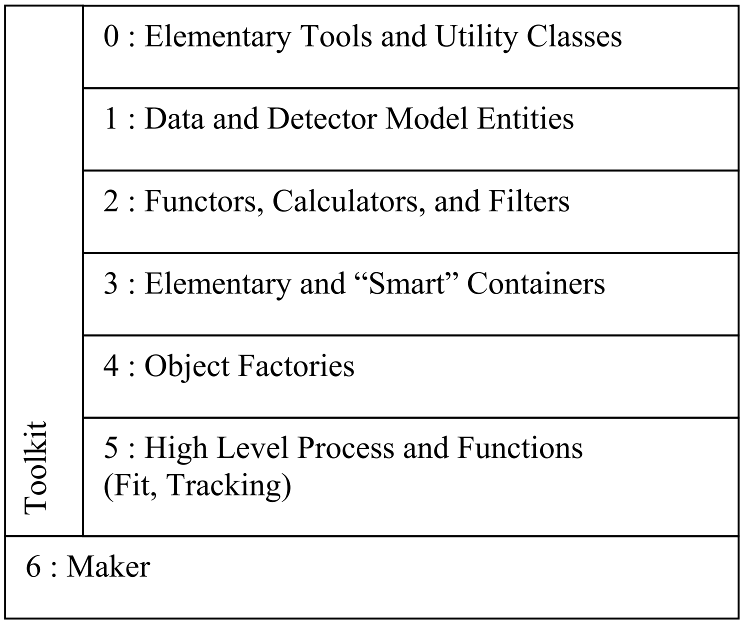

Toolkit
0 : Elementary Tools and Utility Classes
1 : Data and Detector Model Entities
2 : Functors, Calculators, and Filters
3 : Elementary and “Smart” Containers
4 : Object Factories
5 : High Level Process and Functions (Fit, Tracking)
6 : Maker

Figure 4-1 Tier structure of the tracker code. See text for details.

These are discussed in Section. Object factories, used to simply the choice of object type, handle memory allocation, and speed up access to large numbers of "small" objects are described in Section. The actual tracking, and fitting is part of the fifth tier family of classes developed in this project. The key classes are presented in Section . Given the multiple components used in this code, and in view of the complexity of the interdependencies, and relationships, a toolkit was developed to handle the instantiation of the key major components of the system. It is described in section .

4.3.1 Abstract vs. Concrete Classes: Plug-and-play Model

The Sti package contains a mixture of abstract and concrete classes. Although one should ideally first define and separate abstract classes from concrete and work classes, such a level of abstraction was not systematically pursued as it somewhat detracted from the main goal of this project i.e. the development and deployment of a fast and efficiency track reconstruction code in a timely fashion. Yet, the need for easy maintenance, and upgrade-ability prompted us to define a number of abstract interfaces so one could easily substitutes new components to those nominally developed in this project. Examples of such abstract classes include StiTrack, StiTrackFilter, StiTrackFitter, StiTrackSeedFinder., and StiTrackFinder. Pure abstract classes were however not deemed necessary for entities like hits and detector elements, which in this tracker actually are actually quite simple. Classes such as StiHit and StiDetector are not pure virtual, and thus define the accessors, as well as the data members of hits and detector entities.

We recognize that the generalization of this tracker might potentially benefit from a higher degree of abstraction through the use of pure virtual classes, but we nonetheless deem the current package sufficiently general to be usable by many other experiments.

4.3.2 Tools and Utilities

A number of tool and utilities were created to facilitate the development and provide support for high-level operations. A messenging system, described in Section 4.3.2.1 was developed for i/o of debugging, informational and error messages, internally by the code. The system provides multiple concurrent streams whose output level can be set, at run-time, by the user, in order to select what messages, and which components of the system should issue on-screen readable messages.

4.3.2.1 MESSAGING SYSTEM

A large, multi-developer project like ITTF needs a robust & flexible way of passing & filtering messages to the user. Especially during development, it is critical that each developer sees debugging information related to his or her code without being distracted by irrelevant output. A flexible messenging system was thus developed to allow in-code user-settable i/o characteristic for debug, informational, and error messages produced internally. The messaging system consist of 3 classes: MessageType, Messenger, and MessengerBuf.

MessageType : Various types of messages are defined in MessageType.h & MessageType.cxx. Each type of message defined corresponds to exactly one static MessageType object. Each MessageType holds a pointer to the output stream where messages of that type should be sent.

Messenger : The Messenger class takes care of message routing. It is a subclass of ostream. Messengers may be constructed by the user, and each Messenger contains a routing bitmask. Each bit in the mask corresponds to one MessageType, and so the Messenger may send messages corresponding to multiple MessageTypes. There is also a static routing mask in the Messenger class, which indicates which MessageTypes are allowed at all. (All others are ignored.) The instance & static routing masks are ANDed to determine which MessageTypes are output. The usage is is shown in Figure 4-2.

MessengerBuf : MessengerBuf does the low level work of the Messaging system. It subclasses streambuf and overrides streambuf:: overflow(int). This is the method that actually outputs characters to a device or file when the internal buffer is full.

```
// This must be called before using the Messenger system.  It sets the global
// routing mask to allow only output of messages related to hits.
Messenger::init(MessageType::kHitMessage);

// gets the message type object for hit-related messages
MessageType *pHitType = MessageType::getTypeByCode(MessageType::kHitMessage);

// redirects all hit-related output to a file.
pHitType->setOstream(new ofstream("hit.txt"));

// retrieves a messenger which should output to 2 message streams
Messenger& messenger1 = *(Messenger::instance(MessageType::kHitMessage |
                                              MessageType::KTrackMessage);
// retrieves a messenger which should output to 1 message stream
Messenger& messenger2 = *(Messenger::instance(MessageType::kNodeMessage);

// since the global routing mask & the mask for the messenger each contain
// kHitMessage, this outputs to the file "hit.txt"
messenger1 << "hi" << endl;

// since the global routing mask & the mask for the messenger have no overlap,
// this does nothing.
messenger2 << "bye" << endl;

// this should be called when done with the messenger framework.
Mesenger::kill();
```

Figure 4-2 Usage example of the Messenger system.

4.3.3 Data and Geometry Entities

The event reconstruction process involves hits, track candidates, track segments, fully reconstructed tracks, detector elements, etc. These constructs are considered as data and geometry entities given they carry information about the reconstructed data as well as the detector geometry.

4.3.3.1 DETECTOR AND GEOMETRY MODELING

The Sti classes involved in the modeling of the detector include : StiDetector, StiMaterial, StiPlacement, StiShape, StiPlanarShape, StiCylindricalShape, and StiConicalShape. A drawable version of StiDetector is available in the StiGui package. StiDetector is the master geometry entity. It holds pointers to placement, shape, and shape objects, and as such encapsulates all geometry and material notions required for the track reconstruction with a Kalman filter. See Chapter 8 for a detailed description of the detector model and related classes.

4.3.3.2 HIT MODELING

The hit modeling required for track reconstruction purposes, as described in section, is rather simple. It is implemented through the use of a single class called StiHit. No explicit detector type distinctions are made, although each instance of the StiHit holds a pointer reference to a detector object. See Chapter 6 for a detailed description of the hit model and related classes.

4.3.3.3 TRACK MODELING

The abstract class StiTrack defines the notion of track and provides abstract (pure virtual) methods to set and get the properties of the track. At variance with the StEvent/StTrack model, we have opted to have the tracks handle all their properties without the help of an ancillary track model class. It is thus possible to query a track objects for its kinematical properties such the 3-momentum, the transverse momentum, the rapidity, as well as hit information.

Two concrete classes, StiKalmanTrack and StiMcTrack are used to handle reconstructed and Monte Carlo tracks respectively. They are both derived from the StiTrack abstract base class. The StiKalmanTrack class has been designed to accommodate a complex track model, one in which tracks may consist of simple sequences of points, or more complex tree-like structures allowing multi-paths searches starting with a common trunk. The StiKalmanTrack represents the tracks internally using StiKalmanTrackNode instances discussed below. A given StiKalmanTrack instance, a track, holds a pointer to the first and last nodes on the track, and can access through these all other node/hits on the track, thereby obtaining track parameters such as the Kalman state vector (see Section), the momentum, rapidity, etc. The StiKalmanTrack implements “fit()” and “find()” methods that delegate the search (meaning extension of the existing track) and fitting of a track using a delegate pattern. An instance of the StiKalmanTrackFinder class is called to carry the track search and extension, while the “fit()” method delegates the fit to the StiKalmanFit class. Note that both these classes, as well as the StiKalmanTrack class further delgate many operation to the StiKalmanTrackNode class which carries the actual data about the track at any given point on the track.

The StiMcTrack class was to define to enable operation on Monte Carlo tracks internally within the Sti environment. This class is useful in particular for use in combination with the event display where Monte Carlo and and reconstructed tracks can be overlaid thereby permitting tuning or further debugging or development of this tracker. The StiMcTrack is derived from the StiTrack class and thus enables polymorph use of StiTrack containers and operations. In order to avoid a full deployment of a Monte Carlo class, the StiMcTrack class is implemented as a wrapper class around the StMcEvent/StMcTrack class. It uses the “façade” pattern to provide access to all the parameters of the actual StMcTrack but with an interface which common to StiKalmanTrack instances.

An abstract base class StiDrawableTrack is defined to provide drawable functionality. Concrete classes StiRootDrawableKalmanTrack and StiRootDrawableMcTrack are derived simultaneously from the StiDrawableTrack and respectively the StiKalmanTrack and StiMcTrack classes to implement drawable Kalman and Monte Carlo tracks.

The StiKalmanTrackNode class is the workhorse of this tracker. An instance of this class carries a representation of the track at a given position. As such its data members include the “x” position of the node, the Kalman state of the track at this node (see Section for a definition of the track model), a pointer to a hit (possibly null if the track node is within a non active volume acting as a scattering center), a pointer to a parent node, a vector of pointers to children. A node without a parent is the first hit/node on a track. A node without children is the last node on a track. Currently the implementation of the tracker allowes for only one child at each node, but the use of an STL vector to carry the children nodes shall allow an extension of this tracker where a tree-like track model is used.

Simple tasks such as track filtering, the calculation of hit errors, or track energy loss, are delegated to helper “specialty” classes such as StiFilter, StiDedxCalculator, etc. Some of these specialty calculations are implement in fully articulated classes (e.g. StiDedxCalculator), other necessitate simple functors only (HitErrorCalculartor).

A detailed descripton of the track model and related classes can be found in Chapter 7.

5 Basic Classes

We have defined a number of basic classes as a foundation to the Sti code. The most important of these classes are described briefly in the following sections.

5.1 Named and Described

Many objects (but not all) objects need to be labeled with a name and given a description. We have thus implemented two classes Named, and Described to provide this functionality. The two classes have similar definitions and implementations. Only the Named class is discussed here.

The definition of the class is shown in . First note that the constructor of the class is declared protected. One thus cannot instantiate this class on its own. Its use is reserved as a base class to other classes to provide the name attribute. The name is stored as STD string _name. The name can be accessed and set with the getName and setName methods. The isName and isNamedAs methods return true if the name of the object is equal to that given as an argument. The isNamed method returns true if the name is set to a non empty string.

```
class Named
 {
public:
virtual ~Named();
// Set the name of the object
void setName(const string & newName);
/// Get the name of the object
const string getName() const;
/// Determine whether name is set, i.e object has a name
bool isNamed() const;
/// Determine whether name equals given name
bool isName(const string & aName) const;
/// Determine whether name equals that of given object
bool isNamedAs(const Named & named) const;
protected:
/// Only derived class are Named
Named(const string & aName=" ");
string _name;
};
```

Figure 5-1 Definition of the class Named.

5.2 Factory

The notion of object factory was implemented to enable use of runtime selectable class types. This is particularly valuable in the context of the tracker event display where additional features are needed relative to those needed in the basic tracker. The notion of factory is also used to provide a transparent source of objects which are internally managed. (See the description of the VectorizedFactory class.). A call to the getInstance of a given factory returns an instance of the class served by the factory. This is in essence equivalent to a call to the new operator. However, the logistics of the memory management is delegated to the factory rather than the calling program. The user does not have control nor should he/she delete the objects served by the factory. The factory mechanism also allows to return instance of derived classes selected at run time depending on the external conditions.

The definition of the class is shown in REF _Ref40851936 \h Figure 4-4. The class is templated with a single template argument. The argument is the (base) class to be served by the factory. The class inherits from the Named class so different factories can be instantiated and given labels or names. This is a pure virtual class: all its methods are pure virtual. As such, it cannot be instantiated on its own.

The initialize method shall be implemented in derived class to provide for initialization of the factory. The reset method shall be implemented in derived classes to reset the factory service. The getInstance method is that which serves objects of the Factorized type. It must also be implemented by the derived class. The destructor of this class performs no operation but it is implicitly assumed that the destructor of derived class have the responsibility to perform the memory management of the objects they serve.

```
template<class Factorized>
class Factory : public Named
{
public:
Factory(const string& name);
virtual ~Factory();
virtual void initialize()=0;
virtual void reset()=0;
virtual Factorized * getInstance()=0;
};
```

Figure 5-2 Definition of the Factory class.

5.3 Vectorized Factory

The tracker uses a very large number of small objects such as hits, tracks, track nodes, etc in the reconstruction of the events. Although C++ provides dynamic memory allocation, it is in practice unwise to constantly construct and destruct thousands of little objects because the malloc operation is time consuming and it is quite easy to end up with dandling objects (i.e. memory leaks). We thus developed a class VectorizedFactory which provides for a one time allocation of large number of small objects of the same class, and take ownership of these objects for memory management purposes. The class uses the Factory pattern and as such enabled to serve objects of deriving from a selected base class. The actual type of the objects served can however be decided at run time.

The definition of the class is provided in REF _Ref40852462 \h Figure 4-5. The class is templated with two template arguments. It inherits from the Factory class to provide the functionality of an object factory. The first template argument is the name of the actual class of objects to be served. The second argument is the base class of objects to be served. The two arguments may be the same or different. The Concrete class must however always inherit from the Abstract class. This is a concrete class: it can be instantiated with any template arguments to serve objects of that class. A usage example is provided in .

The constructor VectorizedFactory() is private and should not be used. Users of this class must instead use the long constructor VectorizedFactory(const string& name, int original, int incremental, int maxInc) and specify a factory name, the initial number of objects to be instantiated, the incremental number, and the maximum number of times the internal storage may be increased.

The private data members _originalSize, _incrementalSize, _maxIncrementCount, _incrementCount specify respectively the original number of instances requested, the number to be instantiated when a shortage of objects is encountered, the maximum number of such increments, and the current number of increments. Accessors and modifiers to these data members are provided in the class definition. Additionally, the method getCurrentSize returns the current size of the container.

The variable _container is an STL vector of pointers to objects of the base class Abstract used as second template argument of the factory at instantiation. The variable _current is an iterator to the next object instance to be served by the getInstance method.

```
template <class Concrete, class Abstract>
class VectorizedFactory : public Factory<Abstract>
{
public:
VectorizedFactory(const string& newName,
int original,
int incremental,
int maxInc);
virtual ~ VectorizedFactory();
virtual Abstract * getInstance();
void reset();
void initialize();
void setIncrementalSize(int);
void setMaxIncrementCount(int);
int getIncrementalSize() const;
int getMaxIncrementCount() const;
int getCurrentSize() const;
static const int defaultMaxIncrementCount; // 10
   static const int defaultIncrementSize;     // 5000
   static const int defaultOriginalSize;      // 100000
protected:
typedef vector<Abstract*> t_vector;
virtual void instantiate(int);
void destroy();
private:
VectorizedFactory(); //Not implemented
int _originalSize;
int _incrementalSize;
int _maxIncrementCount;
int _incrementCount;
t_vector _container;
t_vector::iterator _current;
};
```

Equation 5-1 Definition of the VectorizedFactory class.

The getInstance method is the work horse of this class. Each call to this method returns the next available object held by the factory. If objects currently held are exhausted, i.e. all served, than a call is internally generated to the protected method instantiate(_incrementalSize) to instantiate (by actually calling the new operator) the given number of objects. The class internally keeps track of the number of times calls to this instantiate method are effected. An exception is thrown if the maximum set at factory construction time is exceeded.

The method reset() resets the internal iterator to the beginning of the _container vector.

In the tracker, factories are used for hits (StiHit), tracks (StiKalmanTrack and StiMcTrack), track nodes (StiKalmanTrackNode), etc, etc. For analysis and reconstuction of heavy ion collisions, the factories are initialized to contain rather large number of elements. The elements are used once per collision analyzed. That is, one resets all factorized once per collision analyzed – no more, no less.

The tracking code invokes the factories to get instances of the relevant objects but does not need to manage them. The delete operator is never called by the tracker classes. Deletion of objects created by the factories is performed at the very end of the execution of the tracker code, when the factories themselves are destructed: the object deletion is carried in the destructor of the VectorizedFactory class.

```
VectorizedFactory<StiKalmanTrack,StiTrack> * factory;
factory = new
VectorizedFactory<StiKalmanTrack,StiTrack>("KalmanTrackFactory",2000,1000,5);
StiTrack * track;
for (int I = 0; I<200;I++)
{
track = factory->getInstance();
//do something with the track
}
factory->reset();
// now ready to start all over…
```

Figure 5-3 Example of the use of the VectorizedFactory class.

5.4 Parameter, ConstrainedParameter, and EditableParameter

The parameter class was developed to provide the notion of named parameter or variable. The definition of the Parameter class, located in the Sti/Base package, is shown in .

```
class Parameter : public Named, public Described
{
public:
static const int Boolean;
static const int Integer;
static const int Float;
static const int Double;
Parameter();
Parameter(const string & name, const string & description, double value,
int type, int key);
Parameter(const string & name, const string & description, bool  * value,
int key);
Parameter(const string & name, const string & description, int   * value,
int key);
Parameter(const string & name, const string & description, float * value,
int key);
Parameter(const string & name, const string & description, double* value,
int key);
Parameter(const Parameter & parameter);
virtual ~Parameter();
const Parameter & operator=(const Parameter & parameter);
  int    getKey() const;
  int    getType() const;
  bool   getBoolValue() const;
  int    getIntValue() const;
float  getFloatValue() const;
double getDoubleValue() const;
  void   setKey(int key);
  void   setValue(bool value);
  void   setValue(int  value);
  void   setValue(float  value);
  void   setValue(double  value);
  void      set(const  string  &  name,const  string  &  description,  double
value,int type=Double,int key=0);
  void      set(const  string  &  name,const  string  &  description,  bool    *
value,int key=0);
  void   set(const string & name,const string & description, int   * value,int
key=0);
  void   set(const string & name,const string & description, float * value,int
key=0);
  void      set(const  string  &  name,const  string  &  description,  double*
value,int key=0);
protected:
  int     _key;
  int     _type;
double  _value;
void *  _exValue;
};
```

Figure 5-4 Definition of the Parameter class.

The Parameter class is a concrete class inheriting from the Named and Described classes. Instances of this class can thus be labelled with a name and given a short description. Object of this class may be use to encapsulate Boolean, Integer, Float, or Double values.

The class features four protected data members. The _key variable is a user defined integer index. It can be used as an indexer, or an actual key. (See example in the context of the StiTrack class). The _type is an integer set to one the four values const int Boolean, const int Integer, const int Float, const int Double defined as static public members of this class. The variable _value normally holds the value of the Parameter object. This behavior may be altered to use the external storage pointed at by the _exValue pointer. The class provides self explanatory accessor and modifiers to these protected data members.

The getValue and setValue normally get and set the value held by the variable _value. This is behavior changes if the _exValue pointer is non null. The value set or returned in that case, is that pointed at by the _exValue variable.

Many constructors are provided to suit varying needs and applications of this class. The parameterless constructor can be used to instantiate an empty and nameless object. The long constructor Parameter(const string & name, const string & description, double value, int type, int key) should be used to construct and initialize a Parameter object with the given name, description, value, type, and key. The four other long constructor may be used to construct and initialize objects on the basis of the value pointed at by the supplied pointer. In this case, the class uses the variable _exValue rather than _variable to carry the Parameter value. This is useful when Parameter objects are embeded as a member of another class but used as interface to data members of that class.

The class ConstrainedParameter and EditableParameter add to the functionality of the Parameter to provide boundaries (low and high) as well as GUI editor features. Their class definitions are presented in respectively.

```
class ConstrainedParameter : public Parameter
{
public:
ConstrainedParameter();
ConstrainedParameter(const string & name, const string & description,double value,
double defaultValue,
double min, double max, int type=Double, int key=0);
ConstrainedParameter(const string & name, const string & description,
bool * value, bool defaultValue, int key=0);
ConstrainedParameter(const string & name, const string & description,
int *  value, int    defaultValue, int min, int max, int key=0);
ConstrainedParameter(const string & name, const string & description,
float* value, float  defaultValue,float min, float max, int key=0);
ConstrainedParameter(const string & name, const string & description,
double *value, double defaultValue, double min, double max,int key=0);
ConstrainedParameter(const ConstrainedParameter & parameter);
virtual ~ConstrainedParameter();
const ConstrainedParameter & operator=(const ConstrainedParameter & parameter);
double  getMinimum() const;
double  getMaximum() const;
double  getDefault() const;
  void    setMinimum(double min);
  void    setMaximum(double max);
  void    setDefault(double value);
  void    setValue(double value);
  void    set(const string & name,const string & description,
            double   value,   double   defaultValue,   double   min,   double   max,   int
type=Double,
            int key=0);
  void    set(const string & name,const string & description,
            bool*value, bool defaultValue, int key=0);
  void    set(const string & name,const string & description,
            int *value, int  defaultValue, int min,  int max, int key=0);
  void    set(const string & name,const string & description,
            float*value, float defaultValue, float min, float max, int key=0);
  void    set(const string & name,const string & description,
            double*value, double defaultValue, double min, double max, int key=0);
  void    reset();
protected:
double  _minimum;
double  _maximum;
double  _default;
};
```

Figure 5-5 Definition of the ConstrainedParameter class.

The ConstrainedParameter class inherits from the Parameter class with public acces. Object of this class thus inherit all properties of the Parameter class. The ConstrainedParameter additonally feature a _minumum, and a _maximum value not to be exceeded. These constrained are used by the setValue methods to limit the range of settable values. The class also features a _default variable which is used to hold the default value of the parameter.

The EditableParameter class inherits with public access from the ConstrainedParameter class. As such it thus has all attributes of this class as well as those of the Parameter base class. It adds the _increment attribute that can be used in a graphical user interface display to determine automatic increment to the Parameter value edited on display.

The definition and implementation of the ConstrainedParameter and EditableParameter are rather similar in spirit to that of the Parameter base class.

```
class EditableParameter : public ConstrainedParameter
{
```

```
public:
EditableParameter();
EditableParameter(const string & name,
const string & description,
double value,
double defaultValue,
double min,
double max,
double increment,
                        int    type,
                        int    key);
EditableParameter(const string & name,
const string & description,
bool*  value,
bool  defaultValue,
int   key);
EditableParameter(const string & name,
const string & description,
                        int*   value,
                        int   defaultValue,
                        int   min,
                        int   max,
                        int   increment,
                        int   key);
EditableParameter(const string & name,
const string & description,
float* value,
float defaultValue,
float min,
float max,
float increment,
int    key);
EditableParameter(const string & name,
const string & description,
double* value,
double defaultValue,
double min,
double max,
double increment,
int    key);
EditableParameter(const EditableParameter & parameter);
virtual ~EditableParameter();
const EditableParameter & operator=(const EditableParameter & parameter);
double  getIncrement() const;
void    setIncrement(double increment);
EditableParameter* set(const string & name, const string & description,
double value, double defaultValue, double min, double max, double increment, int
type,int    key);
EditableParameter* set(const string & name,const string & description,bool*value,
bool defaultValue, int key=0);
EditableParameter* set(const string & name,const string & description,int *value, int
defaultValue, int min,  int max, int increment,int key=0);
EditableParameter* set(const string & name,const string & description,float*value,
float defaultValue, float min, float max, float increment,int key=0);
EditableParameter* set(const string & name,const string & description,double*value,
double defaultValue, double min, double max, double increment,int key=0);
void    reset();
friend ostream& operator<<(ostream& os, const EditableParameter&par);
protected:
double  _increment;
};
```

Figure 5-6 Definition of the EditableParameter class.

5.5 EditableParameters

The class EditableParameters essentially behaves as an STL vector of instances of the Parameter class. It inherits from the Named class. Instances of this class can therefore be given a name. It also inherits from the Subject class. It can then be used where the Observer/Subject pattern is applicable. Instances of this class and of derived classes are used through out the tracker code to provide for graphical user interface editble parameters.

```
class EditableParameters : public Parameters, public Subject
{
public:
EditableParameters();
EditableParameters(const string & name, const string & description);
EditableParameters(const EditableParameters & parameter);
virtual ~EditableParameters();
const EditableParameters & operator=(const EditableParameters & parameter);
virtual void setDefaults();
friend ostream& operator<<(ostream& os, const EditableParameters&pars);
};
```

Figure 5-7 Definition of the EditableParameter class.

5.6 Abstract Filters

Hit and track filters were developed for the dual purposes of track finding and event display. Tracks produced by the seed finder and the Kalman Track finder must be filtered to eliminate low quality and irrelevant tracks. Hit and track filtering are used with the event display to select the characteristics of hits and tracks to be shown as well as for comparative analysis of reconstructed and Monte Carlo tracks.

Filters are based on a pure virtual templated class named Filter. The definition of the class is shown in Figure 5-8.

```
template<class Filtered> class Filter
{
public:
Filter();
virtual ~Filter();
virtual bool accept(const Filtered *filtered) const=0;
virtual void reset();
bool filter(const Filtered * filtered);
int  getAnalyzedCount();
int  getAcceptedCount();
protected:
int _analyzedCount;
int _acceptedCount;
};
```

Figure 5-8 Definition of the pure virtual templated class Filter

The filter class is defined as templated class with a template argument set to be the class of the object to be filtered: e.g. StiHit and StiTrack classes. This enables filtering on virtually any kind of object. Although, it is currently used for tracks, and hits, the construct can be used for many other classes such as events, or vos, etc.

The class is pure virtual because the actual implementation of the filtering depends on the specificities of the objects to be filtered. The class nonetheless provides for a few features common to all single object filters. Two counters are provided as protected data members _analyzedCount and _acceptedCount to keep track of the number of objects analyzed and accepted by the filter. Accessor methods getAnalyzedCount and getAcceptedCount are provided to access these two counters. The value of the counters may be initialized to zero with a call to the reset method.

The method filter, whose implementation is shown in should be called by user code to filter objects. It internally calls the accept method and perform the accounting of analyzed and accepted tracks. If the accept method is called directly this accounting will not be performed and the counter values will be meaningless.

The accept method is pure virtual and must as such be implemented by in derived classes inheriting from the Filter class.

```
template<class Filtered>
inline bool Filter<Filtered>::filter(const Filtered *filtered)
{
_analyzedCount++;
bool acc = accept(filtered);
if (acc) _acceptedCount++;
return acc;
}
```

Figure 5-9 Implementation of the filter method.

The notion of track filter is defined in an abstract class called StiTrackFilter. This filter class provides methods and data members to count and keep track of the number of tracks analyzed, and accepted by the filter. User code should call the filter method of the filter which internally calls an "accept(track)" method and keep an account of the number of tracks analyzed and accepted. The StiTrackFilter is pure virtual and does not implement the "accept(track)" method. The implementation is left to derived classes.

The track filtering is a process that must be fast and flexible. One should also be able to change the number and nature of filtering cuts applied dynamically at run-time. Indeed, we often track an event, change the filter criterion, and re-track the event graphically to study the effects of different track cuts. Two competing approached have been used for the implementation of the filter. Both are briefly discussed below.

5.7 Editable Filters

It is often convenient to use the event display graphical user interface to set the filter parameter values in order to study or debug the tracker. An editable filter class was thus created to provide such a functionality. It is called EditableFilter and it is part of the Sti/Base package. Similarly to the Filter class, it is defined as a pure virtual abstract templated class. It can thus be used to accommodate filtering of any types of objects.

The class definition is shown in . The class uses double inheritance on the Filter and EditableParameters class to provide filtering and editability.

It implements no method other than the two contractors and destructor, and has no data member but provides access to those publicly inherited from the Filter and EditableParameters classes. The parameterless constructor to define a nameless filter whereas the second constructor may be use to instantiate filters with and a name and a short description.

```
template <class Filtered> class EditableFilter : public Filter<Filtered>, public
EditableParameters
{
public:
EditableFilter();
EditableFilter(const string & name, const string & description);
virtual ~EditableFilter();
};
```

Figure 5-10 Definition of the EditableFilter class.

5.8 Track Filter

The class StiDefaultTrackFilter is provided and implemented as the default track filter used in the Sti code. It is a concrete class which inherit from the EditableFilter class with the concrete class StiTrack as its template value. Instances of type StiDefaultTrackFilter thus act as StiTrack filters and have the added convenience of being editable from the event display.

The class definition is shown in Figure 5-11. The parameterless constructor enables the construction of unnamed filters. The long constructor enables users to name and describe the filter. This is useful for filters used and accessible from a graphical user interface. The destructor performs no operations. The initialize method provides a useful filter initialization. The filter can however also be initialized externally using the "add" method of the EditableParameters class.

```
class StiDefaultTrackFilter : public EditableFilter<StiTrack>
{
public:
StiDefaultTrackFilter();
StiDefaultTrackFilter(const string & name, const string & description);
virtual ~StiDefaultTrackFilter();
bool accept(const StiTrack *filtered) const;
virtual void initialize();
};
```

Figure 5-11 Definition of the class StiDefaultTrackFilter.

The accept method implements the actual track filtering. It is shown in . The filter is setup to accept (or reject) tracks on the basis of a sequence of keyed track parameters identified by the parUse->getKey(). One loops through all registered filtering elements. If the filtering elements are enabled i.e. if parUse->getBoolValue() returns true, one tests by a call to the getValue(key) if the track value is within bounds prescribed by the filter element. The StiTrack class is setup to returned the value of a specific track parameter predicated by the value of the key. The value retrieved from the track is thus deemed acceptable if in between the limits of the filter element.

```
inline bool StiDefaultTrackFilter::accept(const StiTrack * t) const
{
double v,low,high;
ParameterConstIterator iter = begin();
Parameter * parUse;
Parameter * parLow;
Parameter * parHi;
while (iter!=end())
   {
parUse = *iter; iter++;
parLow = *iter; iter++;
parHi  = *iter; iter++;
if (parUse&&parLow&&parHi)
         {
if (parUse->getBoolValue())
            {
v = t->getValue(parUse->getKey());
low = parLow->getDoubleValue();
high = parHi->getDoubleValue();
if (v<low || v>high) return false;
            }
            }
    }
return true;
}
```

Figure 5-12 Implementation of the accept() method used for track filtering.

6 Hits

6.1 Hit Representation

The representation of hits is accomplished with the class StiHit. As discussed in details in Chapter 3, track finding and fitting proceeds in coordinates local to the detector where the hits are found. The class StiHit is thus designed to store the hits in the local detector reference frame. However, because the position in global STAR coordinates are also frequently needed, one also stores internally the global position of the point.

The class definition is shown in Figure 6-1.

6.1.1 Data Members

The class features many protected data members which we describe briefly in the following.

The variable *mrefangle* corresponds to the orientation angle of the detector element the hit is issued from. It is used locally in this class to enable fast access to the orientation of the detector rather than dereferencing the detector pointer.

The *mposition* holds the distance between the detector plane where this hit was measured and the origin.

The three values mx, my, and mz and the coordinates of the hit in the local detector reference frame.

The six variables msxx, msyy, mszz, msxy, msxz, and msyz are the error matrix of the hit measurment.

The three values _xg, _yg, and _zg are the coordinates of the hit in global STAR coordinates.

The vairable mTimesUsed is a counter to keep track of the number of times each hit is used. Note that in the current version of the tracker, hits may not be shared by tracks or reused once they have been reserved, i.e. once this counter equals 1. Two values are thus allowed for counter: zero and one.

The variable mdetector is a pointer to the StiDetector object representing the detector where the hit was measured.

The variable msthit is a pointer to the StHit object carrying the StEvent version of the hit.

The variable _energy holds the energy deposition or loss associated with this hit.

```
class StiHit
{
public:
  enum StiHitProperty {kR,kZ,kPseudoRapidity,kPhi};
    StiHit();
    StiHit(const StiHit&);
    const StiHit& operator=(const StiHit& );
    ~StiHit();
    ///Return the local x value.
    float x() const;
```

```
///Return the local y value.
float y() const;
///Return the local/global z value.
float z() const;
///Return the local x value.
float x_g() const;
///Return the local y value.
float y_g() const;
///Return the local/global z value.
float z_g() const;
///Return the (x,x) component of the error matrix.
float sxx() const;
///Return the (y,y) component of the error matrix.
float syy() const;
///Return the (z,z) component of the error matrix.
float szz() const;
///Return the (x,y) component of the error matrix.
float sxy() const;
///Return the (x,z) component of the error matrix.
float sxz() const;
///Return the (y,z) component of the error matrix.
float syz() const;
///Return the energy deposition associated with this point
float getEloss();
///Return the refAngle of the detector plane from which the hit
///arose.
float refangle() const;
///Return the position of the detector plane from whcih the hit
///arose.
float position() const;
///Return a const pointer to the StiDetector object from which the
///hit arose.
const StiDetector* detector() const;
///Return a const pointer to the StHit object corresponding to this
///StiHit instance
//const StHit* stHit() const;
const StMeasuredPoint * stHit() const;
///Return the number of times this hit was assigned to a track
unsigned int timesUsed() const;
///Return a const reference to a StThreeVectorF that denotes the
///position of the hit in global STAR coordinates.
const StThreeVectorF globalPosition() const;
void set(float position,  float angle, float y, float z);
///Set the local position and error in one function call
void set(const StiDetector * detector,
            const StMeasuredPoint * stHit,
            float energy,
            float x, float y, float z,
            float sxx=1, float sxy=1, float sxz=1, float syy=1, float syz=1, float szz=1);
///Set the global position and error in one function call
///A transformation is performed internally from global to local coordinates
///according to the detector information.
void setGlobal(const StiDetector * detector,
                    const StMeasuredPoint * stHit,
                    float x, float y, float z,
                    float energy);
```

```
    ///Set the position error matrix for the measurement from an
    ///StMatrixF object.
    void setError(const StMatrixF&);
    ///Scale all errors by the given factor
    void scaleError(float);
    ///Set the pointer to the StiDetector from which the hit arose.
    void setDetector(const StiDetector*det) {mdetector=det;};
    ///Set the pointer to the corresponding StHit object.
    void setStHit(const StMeasuredPoint*hit){msthit=hit;}
    ///Set the number of times used
    void setTimesUsed(unsigned int);
    void reset();
    void rotate(double angle);
    double getValue(int key) const;
    double getPseudoRapidity() const;
    friend ostream& operator<<(ostream& os, const StiHit& h);
private:
                    float mrefangle;
    float mposition;
    float mx;
    float my;
    float mz;
    float msxx;
    float msyy;
    float mszz;
    float msxy;
    float msxz;
    float msyz;
    // global position
    float _xg,_yg,_zg;
    unsigned int mTimesUsed;
    const StiDetector* mdetector;
    const StMeasuredPoint * msthit;
    float _energy;
};
```

Figure 6-1 Definition of the StiHit class.

6.1.2 Public Methods

The class features accessor (get) and modifier (set) methods for all protected data members. Only the non trivial methods are discussed.

void setGlobal(const StiDetector * detector, const StMeasuredPoint * stHit, float x, float y, float z, float energy);

This method is used to initialize a hit on the basis of external data e.g. a StHit. Pointers to the detector where the hit was measured, the StHit used to generate this information must be supplied as the two first arguments. The triplet x,y,z is the position of the hit expressed in STAR global coordinates. It is translated internally into local coordinates. The energy is the energy deposition or loss associated with the hit.

void rotate(double angle);

This method transform the hit coordinates by effecting a rotation by the given angle around the z-axis.

double getValue(int key) const;

This methods returns the hit information identified by the given key. Currently defined key values are kR, kZ, kPseudoRapidity, kPhi corresponding respectively to the radius of the point (distance to the origin), the z or longitudinal position, the pseudorapidity of the point, and the azimuthal angle of the point. This method is useful in combination with the hit filter.

double getPseudoRapidity() const;

This method returns the pseudo-rapidity of the point relative to the z-axis.

6.2 Hit Factory

The ITTF code uses factories for the instantiation of small objects such as hits rather than explicit calls to the C++ new operator. Factories are described in Section 5.2. Hits can be obtained from a factory of type Factory<StiHit>. Such a factory is defined by default in the ITTF tracker and can be obtained from the Sti toolkit (See Chapter 11). StiHit instances may then be recursively obtained by repeatedly calling the getInstance method of the factory as shown in the code example presented in Figure 6-2 below.

```
Factory<StiHit> * hitFactory = StiToolkit::instance()->getHitFactory();
StiHit * hit;
For (int I=0;I<1000;I++)
{
  hit = hitFactory->getInstance();
  // now do something with the hit…
}
```

Figure 6-2 How to use the hit factory to obtain StiHit instances.

In this example, the construct StiToolkit::instance() is used to obtain a pointer to the toolkit singleton. The pointer is then used with the getHitFactory() method to obtain the actual hit factory. The factory is then polled 1000 times to obtain 1000 different hit instances. The user code must NOT call the delete operator and there shall be no memory leak because the memory management is taken care of by the factory.

6.3 Hit Container

The STAR detector features many tracking detectors. The TPC, SVT, SSD, and FTPC all produce hits that be used for track reconstruction purposes. One can even envision using hits from the EMC showermax detector in tracking. We have thus design the hit class, described in the previous section, to be able to handle hits from any detector. A STAR event may thus involve hits from one or many different detectors. One then has two issues to deal with: (1) The number of hits to store, and handle is extremely large in central heavy ion collisions -can be in excess of 100000 – and (2) One should be able to find and retrieve any individual or collection of hits rapidly. We have thus designed and developed a special hit container class, which provides hit storage and sorting. The class is called StiHitContainer. Its definition is shown in Figure 6-3.

StiHitContainer is exactly that—a container for StiHits! Because of the sheer number of hits in a STAR event, StiHitContainer is designed to provide efficient access to a subset of hits that are within a user specified volume. This is accomplished by mapping between a two dimensional key and an STL container of hits. This mapping is described in further detail below.

There is a natural connection between the hits and the detector from which came. However, for implementation purposes, it is convenient to keep these two entities (HitContainer and detectorContainer) separate, but keep a well defined method of communication between the two. In such a way one can maintain a HitContainer and DetectorContainer that can exist in different representations. That is, one can have a hit container that is well behaved in local coordinates which map very naturally to the detector model as well as a HitContainer that can behave naturally in global coordinates, which do not map naturally to the detector model.

The coordinates of the hits stored are described in StiHit. It should be noted that the ITTF project treats the STAR TPC as if it were 12 sectors, each extending to +-200 cm. That is, we map hits from sectors 13-24 to a coordinate system defined by sectors 1-12. That way we do not need to make any distinction between hits that come from different sides of the TPC central membrane. This is motivated by the fact that although the east and west sectors mark a clear distinction in data taking, this distinction is unimportant in pattern recognition.

First we describe the storage of the hits. StiHitContainer treats the hits from a common detector plane (e.g., TPC padrow 13, sector 12) as a sorted std::vector<StiHit*>. The hits are sorted via the functor StizHitLessThan. This functor has a binary predicate that orders hits in a strict less than ordering based upon global z value (see above). Next, StiHitContainer stores these hit-vectors in a std::map<HitMapKey, std::vector<StiHit*>, MapKeyLessThan >.

HitMapKey is a simple struct that stores two values: refAngle and position. MapKeyLessThan is a simple struct that defines a stric less-than ordering for HitMapKey objects. The values refAngle and position are members of the StiHit class. By specifying a HitMapKey, then, one achieves extremely efficient retrieval of the hit-vector for a given detector plane.

Next we discuss the retrieval of hits from the container. As stated above, one can gain access to a hit-vector for a given detector plane by specifying the position and refAngle of a detector (see method hits(double,double). Additionally, one can access the hit-map itself (or at least a const reference to it!) via the method hits(). However, as stated before, StiHitContainer is capable of efficient retrieval of a subset of the hits in an event. This subset can be defined as a subset of the hits from a given detector plane. Perhaps it is easier to elucidate via an example. Suppose one is interested in the hits corresponding to TPC sector 12, padrow 13. Then, this sector/padrow combination can be easily mapped to a position and refAngle. In this detector one can always specify a 'local' coordinate system where any hit is then fully described by two numbers: local y and z (see StiHit for more inforamtion), where y is the distance along the plane (padrow) and z is the global z. Now, suppose one is interested in hits that are within some volume centered at (y0,z0) and bounded by +-deltaD in y and +-deltaZ in z (deltaD is now poorly named—it was meant to represent distance D along a plane). To retrieve the hits from this volume one must call the setDeltaD() and setDeltaZ() methods to establish the bounds. Then one must call one of the setRefPoint() methods. After this, the container has selected the hits within the specified volume, and they can be retreived via the iterator like interface specified by hasMore() and getHit().

StiHitContainer must be cleared, filled, and sorted for each event. A manual call to sortHits() is necessary to achieve the most efficient container implementation.

The StiToolkit maintains two instances of the StiHitContainer to store reconstructed and monte carlo hits. These can be retrieved with calls to the *StiToolkit::getHitContainer()* and *StiToolkit::getMcHitContainer()* methods respectively.

```
class StiHitContainer : public Named, public Described
{
public:
StiHitContainer(const string & name, const string & description);
 virtual ~StiHitContainer();
 ///Set the half-width of the search window in distance along the pad.
 void setDeltaD(double);
 ///Set the half-width of the search window in distance along global z.
 void setDeltaZ(double);
 ///Return the value of deltaD (cm).
 double deltaD() const;
 ///Return the value of deltaZ (cm).
 double deltaZ() const;
 ///Provide for drawable derived class(es?)
 virtual void update();
 //STL wrappers
 ///Add a hit to the container.
 virtual void push_back(StiHit*);
 ///Return the total number of hits in the container.
 virtual unsigned int size() const;
 ///Declare all hits as unused.
 virtual void reset();
 ///Clear all hits from the container.
 virtual void clear();
 ///Sort all of the hits in the container.
 virtual void sortHits();
 ///Ignore hits marked as used (std::stable_partition)
 void partitionUsedHits();
 ///Return a const reference to a vector of hits.
 const HitVectorType& hits(double refangle, double position);
 ///Return a reference to the vectore of hits, or iterators marking
 ///its bounds
 HitVectorType& hits(const StiDetector*);
 HitVectorType::iterator hitsBegin(const StiDetector*);
 HitVectorType::iterator hitsEnd(const StiDetector*);
 ///Get all hits into a simple vector
 HitVectorType getAllHits();
 ///Return a const reference ot the hit-vector map.
 const HitMapToVectorAndEndType& hits() const;
 HitMapToVectorAndEndType& hits();
 ///Set the reference point to define sub-volume of hits to be
 ///accessed.
 void setRefPoint(StiHit* ref);
 ///Set the reference point to define sub-volume of hits to be
 ///accessed.
 void setRefPoint(double position,double refAngle,double y,double z);
 ///Set reference point to be the location of the given node
 void setRefPoint(StiKalmanTrackNode &);
 ///Return a boolean that reflects whether there are more hits
 ///available in the specified sub-volume.
 bool hasMore() const;
 ///Return a pointer to the StiHit object currently pointed to from the
 ///specified sub-volume.
 StiHit* getCurrentHit(); //get current
 ///Return a pointer to the StiHit object currently pointed to from the
 ///specified sub-volume.
 StiHit* getHit();  //get current hit and increment
 ///Return the number of hits satisfying the current reference and
 ///search domain.
 int getHitCandidateCount() const;
 ///Add a vertex to the hit-container.
 void addVertex(StiHit*); //push_back
 ///Return the number of vertices stored in the container.
 unsigned int numberOfVertices() const;
 ///Return a const reference to the a vector of vertices.
 const HitVectorType& vertices() const;
protected:
 Messenger& mMessenger;
private:
 //Vertex implementation
 HitVectorType mvertexvec; //! Container for primary vertices
 ///store a map key member to avoid constructor call per hit
 HitMapToVectorAndEndType::key_type mkey;
 double mdeltad; //Search limit in local d-direction
 double mdeltaz; //Search limit in local z-direction
 //Used to search for points that satisfy users query
 StiHit* mminpoint;
 StiHit* mmaxpoint;
 StiHit mUtilityHit;
 HitVectorType::iterator mstart;
 HitVectorType::iterator mstop;
 //Used to return points that satisfy users query
 HitVectorType::const_iterator mcurrent;
 HitVectorType mcandidatevec;
 HitMapToVectorAndEndType mmap; //! the hit container
};
```

Figure 6-3 Definition of the StiHitContainer class.

6.4 Hit Loading

The track reconstruction process is based on hits stored in an instance of the StiHitContainer. We stress that this container is used rather than say StEvent because it is designed to store hits from arbitrary detectors within a unique container and provides for fast retrieveal of any hit or collection of hits on the basis of a short key. With StEvent, one must loop through complicated and detector dependent containers to retrieve hits, and there are no mechanism to retrieve hits on the basis of a key. Note that it is also possible to switch from real reconstructed hits to MonteCarlo hits – something that simply cannot be done within the exiting tracker…

Currently, the STAR software is setup to fill hits into StEvent. To use the tracker, one must then first extract and copy the information into an appropriate instance of the StiHitContainer class. The StiHitContainer is described in the previous section. Here, we discuss the loading of the container from an external source.

The tracker was designed to enable hit loading from an arbitrary source – StEvent, StMcEvent, or even STAFF tables. This is accomplished through an abstract template event loader class called StiHitLoader. The default implementation uses StEvent and StMcEvent as the source of the hits, but it can be easily changed by overriding the existing loaders.

Essentially, one needs to implement a loader for each detector used in the tracker. A master loader is then used to loop through all registered hit-loaders to load hits from the various relevant detectors. The implementation of detector specific hit-loaders is described in a different section. Here, we describe the abstract base class and the master hit loader.

6.4.1 StiHitLoader Class

The properties and behavior of hit loaders are defined by the abstract interface class named StiHitLoader whose definition is presented in Figure 6-4.

The class StiHitLoader is templated to enable hit loading from different and arbitrary sources. Also note that the class inherits from the Named class thereby allowing for loaders to have a name.

The constructor takes five arguments as follows: name of the loader, pointers to hit containers where to load hits and monte carlo hits (if any), pointer to an appropriate hit factory, and pointer to a detector builder. The detector builder carries information about the detector geometry so one can associate hits with their appropriate detector object at load time.

```
template <class Source1, class Source2, class Detector>
class StiHitLoader : public Named
{
 public:
  StiHitLoader(const string & name,
               StiHitContainer* hitContainer,
               StiHitContainer* mcHitContainer,
               Factory<StiHit>*hitFactory,
                Detector * detector);
  virtual ~StiHitLoader();
  virtual void loadEvent(Source1 *source1,
                         Source2 *source2,
                         Filter<StiTrack> * trackFilter,
                         Filter<StiHit>   * hitFilter);
  virtual void loadHits(Source1 * source,
                        Filter<StiTrack> * trackFilter,
                        Filter<StiHit>   * hitFilter);
  virtual void loadMcHits(Source2 * source,
                          bool useMcAsRec,
                          Filter<StiTrack> * trackFilter,
                          Filter<StiHit>   * hitFilter);
  virtual void setHitContainer(StiHitContainer* hitContainer);
  virtual void setMcHitContainer(StiHitContainer* hitContainer);
  virtual void setHitFactory(Factory<StiHit>*hitFactory);
  virtual void setDetector(Detector*detector);
  virtual Detector* getDetector();
  virtual void setUseMcAsRec(bool value);
  virtual bool useMcAsRec() const;
 protected:
  StiHitContainer     * _hitContainer;
  StiHitContainer     * _mcHitContainer;
  StiTrackContainer   * _trackContainer;
  StiTrackContainer   * _mcTrackContainer;
  Factory<StiHit>     * _hitFactory;
  Factory<StiKalmanTrack> * _trackFactory;
  Factory<StiMcTrack> * _mcTrackFactory;
  Detector            * _detector;
  bool                _useMcAsRec;
};
```

Figure 6-4 Definition of the StiHitLoader class.

The class features the following protected data members:

_hitContainer and *_mcHitContainer* are pointers to respectively reconstructed and monte carlo hit containers. They can be set at construction time with the long class constructor or modified with the setHitContainer() and setMcHitContainer() methods.

_trackContainer and *_mcTrackContainer* are respectively pointers to reconstructed and monte carlo track containers. The StiTrackContainer is described in the next section of this document.

_hitFactory, *_trackFactory*, and *_mcTrackFactory* are pointers to respectively hit factory, track factory of reconstructed tracks (StiKalmanTrack) and factory of monte carlo tracks (StiMcTrack). They are initialized at construction time based on calls to the STI toolkit.

_detector is a pointer to a detector builder. It can be initialized via the class constructor or with the void *setDetector(Detector*detector)* method. The builder is used internally by the loaders attach detector objects to the hits.

_useMcAsRec is a Boolean flag which is set to false in normal track reconstruction mode. A value ==true requests the loader to load monte carlo hits as if they were actual reconstructed hits. This is useful for tracker debugging purposes and it enables to perform track   reconstruction on perfect hits i.e. without detector finite resolution effects.

The class features three action methods “loadEvent”, “loadHits”, and “loadMcHits”. In the context of this base class, only the loadEvent method performs an actual operation. The loadHits and loadMcHits are here implemented with dummy, no ops, methods – it is then unnecessary to implement these in derived classes if one of the other operation are not required.

The implementation of the loadEvent method is shown in Figure 6-5. The method sequentially calls both the loadHit and loadMcHit methods. Note that the calls are effected only if source1 and source2 are valid pointers to StEvent and StMcEvent objects (or other valid source as the case may be). Note also that if the Boolean flag _useMcAsRec is set true, only the monte carlo hits (from StMcEvent) are loaded.

```
template<class Source1, class Source2, class Detector>
void      StiHitLoader<Source1,      Source2,Detector>::loadEvent(Source1
*source1,
                                                   Source2 * source2,
                                            Filter<StiTrack> * trackFilter,
                                                   Filter<StiHit>          *
hitFilter)
{
 cout << “Loader “<<_name<<” loading event”<<endl;
 if (source1 && !_useMcAsRec)
   loadHits(source1,trackFilter,hitFilter);
 if (source2)
   loadMcHits(source2,_useMcAsRec,trackFilter,hitFilter);
}
```

Figure 6-5 Implementation of the loadEvent() method of the StiHitLoader class.

6.4.2 StiMasterHitLoader Class

An instance of the StiMasterHitLoader class is used in the tracker to sequentially call hit loaders appropriate for each detector system (e.g. TPC, SVT, etc).

The class definition is shown in Figure 6-1.The class inherits from the StiHitLoader base class and as such behaves like a hit loader. It also inherit from the STL vector class to behave as a vector of loaders. It thus implements a composite loader which internally use all loaders subscribed to load hits from the given sources.

The StiToolkit holds one instance of a master hit loader. A pointer to the master loader can be retrieved with a call to the StiToolkit::getMasterHitLoader() method. One can then use the addLoader(…) method to add subsystem specific hit loaders to the master loader. For detailed examples, see section on imlpementation of new detectors.

```
template<class Source1, class Source2,class Detector>
class StiMasterHitLoader : public StiHitLoader<Source1,
Source2,Detector>,
                            public vector<
StiHitLoader<Source1, Source2,Detector> *>
{
public:

    StiMasterHitLoader();
    StiMasterHitLoader(const string& name,
                       StiHitContainer* hitContainer,
                       StiHitContainer* mcHitContainer,
                       Factory<StiHit>*hitFactory,
                       Detector*transform);
    virtual ~StiMasterHitLoader();
    void addLoader(StiHitLoader<Source1,
Source2,Detector>*loader);
    void loadEvent(Source1 *source1,
                   Source2 *source2,
                   Filter<StiTrack> * trackFilter,
                   Filter<StiHit>   * hitFilter);
    void setHitContainer(StiHitContainer* hitContainer);
    void setMcHitContainer(StiHitContainer* hitContainer);
    void setHitFactory(Factory<StiHit>*hitFactory);
    virtual void setDetector(Detector*detector);
    virtual void setUseMcAsRec(bool value);
protected:
    typedef StiHitLoader<Source1,Source2,Detector>*
HitLoaderKey;
    typedef vector< HitLoaderKey >  HitLoaderVector;
    typedef HitLoaderVector::iterator HitLoaderIter;
    typedef HitLoaderVector::const_iterator
HitLoaderConstIter;
    //HitLoaderVector _hitLoaders;
};
```

Figure 6-6 Definition of the StiMasterHitLoader class.

The implementatin of the loadEvents() is shown in Figure 6-7. The method proceeds in a loop, using an iterator to call the loadEvent() method of all subsystem hit loaders registered with (added to) the master loader. The hitContainer is then sorted and reset to initialize it properly for use with the track seed finder. Notice that no need for memory management of the hit instance is needed here because this is all handled by the hit factory.

Subsystem or detector hit loader may be registered with the master hit loader by calling the add(StiHitLoader*) of the master loader. This is done for instance in the StiMaker Make method. See documentation on the implementation of detector groups and the deployment of new detector subsystems.

```
template<class Source1, class Source2,class Detector>
void StiMasterHitLoader<Source1, Source2,Detector>::
  loadEvent(Source1 *source1,
            Source2 * source2,
                    Filter<StiTrack> * trackFilter,
                    Filter<StiHit>   * hitFilter)
{
// remove all hits currently in the container.
if(!_hitContainer)
 throw runtime_error("StiMasterHitLoader::loadEvent( ) -F- hitContainer==0");
_hitContainer->clear();
if (source2)
{
 if(!_mcHitContainer)
  throw runtime_error("StiMasterHitLoader::loadEvent( )-F-_hitContainer==0");
 _mcHitContainer->clear();
}
HitLoaderConstIter iter;
for (iter=begin();iter!=end();iter++)
 (*iter)->loadEvent(source1,source2,trackFilter, hitFilter);
_hitContainer->sortHits();
_hitContainer->reset();//declare all hits as unused...
if (source2)
 {
   _mcHitContainer->sortHits();
   _mcHitContainer->reset();
 }
```

Figure 6-7 Implementation of the loadEvent method of the StiMasterHitLoader class.

7 Tracks

The track representation and classes are described in Section 7.1. Track factories used in the tracker are discussed in Section 7.2. The default track filter used in the track reconstruction for quality filtering is described in Section 7.3. The track storage container is described in Section 7.4.

7.1 Track Representation and classes

The notion of tracks can be used in different contexts to represent reconstructed particle trajectories on the basis of measured hits, or a collection of Monte Carlo hits generated by GEANT. Although the two types of tracks have their own source and specificities, they have nonetheless a large number of common attributes. We thus defined an abstract track class or interface from which other track classes used by ITTF must derive. This abstract class, called StiTrack, is summarized in Figure 7-1. The class StiKalmanTrack used in the track reconstruction is described in . The class StiMcTrack used in the event display to represent and display generator tracks is discussed in a different section of this doucument.

7.1.1 Abstract Tracks

The abstract class was designed to provide a simple interface to all attributes commonly needed during track reconstruction and afterward in the analysis on which they are based.

An abreviated view of the class definition is shown in Figure 7-1.

An enumeration StiDirection is first defined to identify the inside-out and outside-in tracking or fitting directions. These values are used within the tracker to perform inside-out and outside-in passes.

An enumeration StiTrackProperty is also defined. The enumeration provides for a list of keys to the various attributes of the track. It is then possible to use the generic getValue method to fetch all attributes of the track on the basis of one simple method call. The mean of the keys is listed in Figure 7-1.

The class defines protected static pointers *StiTrackFinder * trackFinder* and *StiTrackFitter * trackFitter*. These point respectively at the track finder and track fitter objects used at run time. They can be set with the public static methods *void setTrackFinder(StiTrackFinder * finder)* and *void setTrackFitter(StiTrackFitter * fitter)*.

The constructor and destructor of this class perform no operations.

The find and fit methods call the find and fit methods of the finder and fitter pointed at by the static variables *StiTrackFinder * trackFinder* and *StiTrackFitter * trackFitter*. In essense, one can ask a track to find or fit "itself" but it delegates the work to the appropriate track finder and fitters.

All public methods of this class are pure virtual and need to be implemented in derived classes.

The method *reset()* shall be implemented in derived classes to reset or zero the internal storage of the track. This is essential because the track objects are served by a factory. As such they are constantly reused objects. They thus need to be reset before each new use.

The method *void getMomentum(double p[3], double e[6])* shall be implemented in derived classes to return the momentum vector (p[3]) and its error matrix (e[6]).

The methods StThreeVector<double> getMomentumAtOrigin() const;, StThreeVector<double> getMomentumNear(double x);, and StThreeVector<double> getHitPositionNear(double x) const; are pure virtual and need to be implemented in derived classes. The first two shall return the track momentum at the main vertex, and near a position x along the track. The getHitPositionNear shall return the hit position (three vector) closest to the independent variable x.

The methods double getCurvature() const, double getP()const, double getPt()const, double getRapidity()const, double getPseudoRapidity() const, double getPhi() const, double getTanL() const shall all be implemented in derived classes to return the respectively the track curvature, momentum, transverse momentum, rapidity (with the current mass hypothesis assumed by the track), pseudorapidity, azimuthal angle and tangent of the track dip angle (relative to the transverse plane).

```
enum StiDirection {kOutsideIn=0, kInsideOut};

class StiTrack
{
public:

  enum StiTrackProperty {kCharge=0, // charge sign of the track
                         kMass,  // mass hypothesis used
                         kChi2,  // chi2 of the track fit
                         kDca2,  // two-dimension DCA – not implemented
                         kDca3,  // three-dimension DCA – not implemented
                         kFlag,  // place holder
                         kPrimaryDca, // DCA to main vertex
                         kPointCount, // Number of hits on track
                         kFitPointCount, // Number of fit points on track
                         kGapCount,      // Number of empty active layers
                         kTrackLength,   // physical length of track
                         kMaxPointCount, // max number of points on track
                         kisPrimary,  // whether it includes main vertex
                         kTpcDedx,    // dedx value with tpc points
                         kSvtDedx,    // dedx value with svt points
                         kCurvature,  // track curvature
                         kP,          // track momentum
                         kPt,         // track transverse momentum
                         kRapidity,   // track rapidity
                         kPseudoRapidity, // pseudo-rapidity
                         kPhi,  // azimuthal angle relative to x-axis
```

```
                              kTanL  // tan(dip-angle)
                };
  static void setTrackFinder(StiTrackFinder * finder);
  static void setTrackFitter(StiTrackFitter * fitter);
  static StiTrackFinder * getTrackFinder();
  static StiTrackFitter * getTrackFitter();

  StiTrack();
  virtual ~StiTrack();
  virtual void fit(int direction=kOutsideIn);
  virtual bool find(int direction=kOutsideIn);
  virtual void reset()=0;
  virtual void getMomentum(double p[3], double e[6]) const =0;
  virtual StThreeVector<double> getMomentumAtOrigin() const =0;
  virtual StThreeVector<double> getMomentumNear(double x) =0;
  virtual StThreeVector<double> getHitPositionNear(double x) const =0;
  virtual double  getCurvature()      const=0;
  virtual double  getP()              const=0;
  virtual double  getPt()             const=0;
  virtual double  getRapidity()       const=0;
  virtual double  getPseudoRapidity() const=0;
  virtual double  getPhi()            const=0;
  virtual double  getTanL()           const=0;
  virtual int     getPointCount() const=0;
  virtual int     getFitPointCount() const=0;
  virtual int     getGapCount() const=0;
  virtual int     getMaxPointCount() const=0;
  virtual int     getSeedHitCount() const =0;
  virtual void    setSeedHitCount(int c)=0;
  virtual double  getTrackLength() const=0;
  virtual vector<StMeasuredPoint*> stHits() const=0;
  virtual double  getMass() const=0;
  virtual int     getCharge() const=0;
  virtual double  getChi2() const=0;
  virtual void    setFlag(long v)=0;
  virtual long    getFlag() const=0;
  virtual vector<StiHit*> getHits()=0;
  virtual double  getValue(int key) const;
  virtual bool extendToVertex(StiHit* vertex)=0;

 protected:
  static StiTrackFinder * trackFinder;
  static StiTrackFitter * trackFitter;
};
```

Figure 7-1 Definition of the abstract StiTrack class.

The *int getPointCount() const* method shall be implemented in derived classes to return the number of points on the track.

The int *getFitPointCount() const* method shall be implemented in derived classes to return the number of fit points used on the track.

The *int getGapCount() const* method shall be implemented in derived classes to return the number of fit points used on the track.

The *int getMaxPointCount()* const method shall be implemented in derived classes to return the max number of points that can lie on this track given its current geometry.

The *int getSeedHitCount()* const method shall be implemented in derived classes to return the number of seed points used on the track.

The void setSeedHitCount(int c) method shall be implemented in derived classes to set the number of seed points used on the track.

The *double getTrackLength() const* method shall be implemented in derived classes to return the physical length of the track.

The *vector<StMeasuredPoint*> stHits() const* method shall be implemented in derived classes to return the an STL vector of StHit hits associated with this track. This method is called in the StiStEventFiller class to copy StiTracks into the persistent data model StEvent.

The *double getMass() const* method shall be implemented in derived classes to return the mass hypothesis used to fit the track.

The *int getCharge() const* method shall be implemented in derived classes to return the charge of the track.

The *double getChi2() const* method shall be implemented in derived classes to return the chi-square obtained in the fit of the track.

The *void setFlag(long v)* method shall be implemented in derived classes to set a user flag.

The *long getFlag() const;* method shall be implemented in derived classes to return a user flag.

The *vector<StiHit*> getHits();* method shall be implemented in derived classes to return an STL vector of StiHit hits associated with this track.

The *double getValue(int key) const;* method shall be implemented in derived classes to return the track attribute selected by the given key.

The *bool extendToVertex(StiHit* vertex)=0;* method shall be implemented in derived classes to request a physical extension of the track to the given primary vertex position.

7.1.2 Kalman Tracks

The StiKalmanTrack class was designed to provide all attributes and tools necessary in the track reconstruction process. It inherits from the StiTrack interface to provide a simple and common access to all track attributes. It however also implements a number of additional methods required by the tracker.

The class definition is presented in Figure 7-2. The class inherits from the abstract base class StiTrack and implements all of its methods. The definition and interpretation of these methods are discussed in the previous section. They are not repeated here. We instead focus on the methods and data members added in this class.

```
class StiKalmanTrack : public StiTrack
{
 public:
  StiKalmanTrack();
  virtual ~StiKalmanTrack();
  void    reset();
  double  getP() const;
  double  getPt() const;
  double  getCurvature() const;
  double  getRapidity() const;
  /// Return the pseudorapidity of the track.
  double  getPseudoRapidity() const;
  // Return azimuthal angle at inner most point of the track.
  double  getPhi()           const;
  /// Returns the tangent of the dip angle of the track
  /// determined at the inner most point of the track.
  double  getTanL()          const;

 /// Return the number of hits associated with this track.
 int getPointCount() const;
 /// Returns the number of hits associated and used in the
 /// fit of this track.
 int getFitPointCount() const;
 /// Return the number of gaps on this track.
```

```
int getGapCount() const;
double getTrackLength() const;
/*!
Returns the maximum number of points that can possibly be on
the track given its track parameters, i.e. its position in the
detector. The calculation accounts for sublayers that are not active,
and nominally active volumes
that were turned off or had no data for some reason.
*/
int getMaxPointCount() const;
int getSeedHitCount() const;
void setSeedHitCount(int c);
/// Return true if the track contains the main vertex.
 bool isPrimary() const;

double calculateTrackLength() const;
double calculateTrackSegmentLength(const StiKalmanTrackNode &p1, const
                              StiKalmanTrackNode &p2) const;
double getTrackRadLength() const;
int calculatePointCount() const;
int calculateMaxPointCount() const;
double getTpcDedx() const;
double getSvtDedx() const;
StiKTNBidirectionalIterator begin() const;
StiKTNBidirectionalIterator end() const;
/// Accessor method returns the outer most node associated with the
/// track.
StiKalmanTrackNode * getOuterMostNode()  const;
/// Accessor method returns the inner most node associated with the
/// track.
StiKalmanTrackNode * getInnerMostNode()   const;
/// Accessor method returns the outer most hit node associated with
/// the track.
StiKalmanTrackNode * getOuterMostHitNode()  const;
/// Accessor method returns the inner most hit node associated with
/// the track.
StiKalmanTrackNode * getInnerMostHitNode()   const;
/// Accessor method returns the first node associated with the track.
StiKalmanTrackNode * getFirstNode()  const { return firstNode; };
/// Accessor method returns the last node associated with the track.
// Assumes the track has been pruned.
StiKalmanTrackNode * getLastNode()   const { return lastNode; };

void  setLastNode(StiKalmanTrackNode *n) { lastNode = n; };

/// Returns the direction (kInsideOut, kOutsideIn) used in the
///reconstruction of this track.
StiDirection getTrackingDirection() const;
/// Returns the direction (kInsideOut, kOutsideIn) used in the fit of
/// this track.
StiDirection getFittingDirection() const;
/// Sets the direction (kInsideOut, kOutsideIn) used in the
/// reconstruction of this track.
void setTrackingDirection(StiDirection direction);
/// Sets the direction (kInsideOut, kOutsideIn) used in the fit of
/// this track.
void setFittingDirection(StiDirection direction);
/// Method used to add a hit to this track
virtual StiKalmanTrackNode * add(StiHit *h,double alpha, double eta,
                           double curvature, double tanl);

/// Add a kalman track node to this track as a child to the last node
/// of the track and return the added node
virtual StiKalmanTrackNode * add(StiKalmanTrackNode * node);
```

```
/// Work method used to find the node containing the given hit.
StiKalmanTrackNode * findHit(StiHit * h);


/// Convenience method to initialize a track based on seed information
void initialize(double curvature,
                double tanl,
                const StThreeVectorD& origin,
                const HitVectorType &);


 /// Work method returns the node closest to the given position.
  StiKalmanTrackNode *  getNodeNear(double x) const;

  /*! Convenience method returns a point corresponding to the node
    of this track which is the closest to the given position.
  */
  StThreeVector<double> getPointNear(double x) const;
  StThreeVector<double> getGlobalPointNear(double x) const;
  StThreeVector<double> getGlobalPointAt(double x) const;

   StThreeVector<double> getMomentumAtOrigin() const;
   StThreeVector<double> getMomentumNear(double x);
   StThreeVector<double> getHitPositionNear(double x) const;

   virtual vector<StiHit*> getHits();
   virtual vector<StMeasuredPoint*> stHits() const;
   virtual vector<StiKalmanTrackNode*> getNodes(int detectorGroupId)
const;

  // Function to reverse the node geometry of a track
  void swap();
  double  getMass() const;   // mass when pid known
  int     getCharge()const;   // charge of the particle
  double  getChi2() const;   // chi2 of fit
  double  getDca2(StiTrack *t) const;   // distance of closest approach
to given track - 2D calc
  double  getDca3(StiTrack *t) const;   // distance of closest approach
to given track - 3D calc

  bool find(int direction=kOutsideIn);
  void prune();
  void reserveHits();
  bool extendToVertex(StiHit* vertex);

  void setFlag(long v);
  long getFlag() const;

protected:
  static StiKalmanTrackFinderParameters * pars;
  static Factory<StiKalmanTrackNode> * trackNodeFactory;

  StiDirection trackingDirection;
  StiDirection fittingDirection;
  StiKalmanTrackNode * firstNode;
  StiKalmanTrackNode * lastNode;
  int     mSeedHitCount; //number of points used to seed the track
  long    mFlag;         //A flag to pack w/ topo info
  double  m;             // mass hypothesis
  double  _dca;
};
```

Figure 7-2 Definition of the StiKalmanTrack class.

The class holds a protected static pointer *pars* to the tracking parameters specified by an instance of the StiKalmanTrackFinderParameter class.

The class also hods a pointer to a StiKalmanTrackNode factory. The nodes are used internally for tracking. See the conceptual tracking model and the class documentation for details on the defintion and use of the StiKalmanTrackNode class.

The class features variables StiDirection trackingDirection and StiDirection fittingDirection to keep track of the direction used in tracking and in fitting the track i.e. inside-out or outside-in.

The variables *StiKalmanTrackNode * firstNode* and *StiKalmanTrackNode * lastNode* are pointers to the first and last node on the track. Strictly speaking, only the first node is required given that the nodes operate as a link list (actually a tree to be absolutely exact) but having the last node enables faster traversal and track swapping. Public accessor and modifiers methods are provided to externally get and set these pointers.

The methods *StiKTNBidirectionalIterator begin() const* and *StiKTNBidirectionalIterator end() const* return iterators to the beginning and end of the track node list.

The method *StiKalmanTrackNode * getOuterMostNode() const* returns the outer most node associated with the track.

The method *StiKalmanTrackNode * getInnerMostNode()const* returns the inner most node associated with the

The method *StiKalmanTrackNode * getOuterMostHitNode()const* returns the outer most hit node associated with the track.

The method *StiKalmanTrackNode * getInnerMostHitNode() const* returns the inner most hit node associated with the track.

The method *StiKalmanTrackNode * add(StiHit hit,double alpha, double eta, double curvature, double tanl)* is used to add a new hit to the track. Alpha, eta, curvature, and tanl are respectively the reference angle of the detector, Cx_0, the track curvature C, and the tangent of the track dip angle.

The method *StiKalmanTrackNode * add(StiKalmanTrackNode * node)* is used to add a track node to the track and append it after the current last node.

The method *void initialize(double curvature, double tanl, const StThreeVectorD& origin, const HitVectorType &)* is a convenience method used to initialize the track on the basis of a seed consisting of an estimate of the track curvature, tangent of its dip angle, origin, and a vector of StiHit hits.

7.1.3 Monte Carlo Tracks

The Monte Carlo (MC) track class StiMcTrack was defined to enable use of MC tracks hits as a test bed for the tracker. MC tracks can also be loaded and displayed within the event display for debugging, tuning, and public relation purposes.

The definition of the StMcTrack class is shown in Figure 7-3. It implements the abstract class StiTrack. The definition and interpretation of the base class methods are given in Section 7.1.1. They are not repeated here.

```
class StiMcTrack : public StiTrack
{
 public:  StiMcTrack();
  virtual ~StiMcTrack();
  virtual void fit(int direction=kOutsideIn);
  virtual bool find(int direction=kOutsideIn);
  virtual void reset();
  virtual void getMomentum(double p[3], double e[6]) const ;
  virtual StThreeVector<double> getMomentumAtOrigin() const ;
  virtual StThreeVector<double> getMomentumNear(double x) ;
  virtual StThreeVector<double> getHitPositionNear(double x)
const;
  virtual double  getCurvature()      const;
  virtual double  getP()              const;
  virtual double  getPt()             const;   // transverse
momentum
  virtual double  getRapidity()       const;   // rapidity
  virtual double  getPseudoRapidity() const;   // pseudo
rapidity
  virtual double  getPhi()            const;   // azimuthal
angle
  virtual double  getTanL()           const;   // tan(lambda)
  virtual double  getDca()  const;  // DCA to given point/hit
  virtual int     getPointCount() const;
  virtual int     getFitPointCount() const;
  virtual int     getGapCount() const;
  virtual int     getMaxPointCount() const;
  /// number of hits used to seed the track
  virtual int     getSeedHitCount() const ;
  virtual void    setSeedHitCount(int c);
  virtual double  getTrackLength() const;
  virtual vector<StMeasuredPoint*> stHits() const;
  /// Get mass of the particle that produced this track
  virtual double  getMass() const;
  /// Get charge of the particle that produced this track
  virtual int     getCharge() const;
  /// Get chi2 of this track
  virtual double  getChi2() const;
  virtual void    setFlag(long v);
  virtual long    getFlag() const;
  virtual void    setStMcTrack(const StMcTrack * track);
  const StMcTrack * getStMcTrack(const StMcTrack * track)
const;
  bool extendToVertex(StiHit* vertex);
  vector<StiHit*> getHits();
 protected:
  const StMcTrack * mcTrack;
  vector<StiHit*> _hits;
};
```

Figure 7-3 Definition of the StiMcTrack class.

This class derives from the StiTrack. Instances of this class thus behave like StiTrack tracks and can be used wherever StiTrack instances may be used. It is however implemented as a façade to the actual monte carlo track StMcTrack loaded from StMcEvent events. Essentially all methods of this class call the corresponding method (which often have different names) of StMcTrack to extract the relevant information. An example of method implementation is shown in Figure 7-4.

The class features a protected pointer to the parent StMcTrack track for which it is a façade. Public accessor and modofier methods to this pointer are provided.

The class also feature an STL vector of StiHits for convenience while using instances of this class in the event display. The accessor method vector<StiHit*> getHits() return this vector.

```
double  StiMcTrack::getPseudoRapidity() const
{
  return mcTrack->pseudoRapidity();
}
```

Figure 7-4 Example of the façade pattern used in the implementation of the StiMcTrack class.

7.1.4 Kalman Track Nodes

The StiKalmanTrackNode class definition is presented in Figure 7-5. This is by far the most complicated and busy class of the ITTF tracker. It is actually the work horse of the tracker. It actually has the responsibility of propagating the track through layers, calculating and updating the track Kalman state vector and error matrix.

The track model and Kalman machinery are described in Chapter 10 of this document. Here we discuss the functional aspects of the class. Given the class is indeed rather complicated and large, we focus on the essential methods of the class. Comments included in the header file shown in provide in most cases a straightforward interpreation of the methods functionality. Exceptional cases are discussed below.

The class inherits from the StiTrackNode base class which provide the tree, parent, and dauthers relationships between track nodes. (See code for documentation of that class).

The class provides accessor methods which mimic most the accessor methods of the StiTrack class. In truth, the StiKalmanTrack typcially delegate the tasks of returning momenta and other track parameters to either the first or last node of the track. This is required because as the tracks are swam through the detector their state vector may evolve i.e. the curvature, crossing and dip angle may change.

The method *int propagate(StiKalmanTrackNode *p, const StiDetector * tDet)* propagates a track state vector from the given node p, to the given detecor tDet, and store the result in this node.

Internally, the method first determine the type of volume the track must be extrapolated to. The track is then extrapolated with the appropriate algorithm and calculator. Once the extraplated position is obtained, one checks using the "locate" method whether the track extrapolation is actuall within the active volume of the targeted detector or volume. The locate method returns a code which indicates whether the extrapolation is well within the volume, on its boundary, or well outside its boundary. Propagate returns if the extrapolated is outside the active boundary of the volume. The next steps are to propagate the error matrix and to include MCS and energy loss effects by calling respectively the propagateError() and propagateMCS methods.

The method *void nudge()* propagates the track held by the node from its current position to the actual position of the hit held by this node. This accounts for the fact that hits may not exactly lie on the detector plane they are issued from because of corrections such as the ExB correction.

The method *double evaluateChi2(const StiHit *hit)* evaluates the chi2 increment associated with the inclusion of the given hit to the track encapsulated by this node.

The method *void updateNode()* updates the Kalman state of the track using the hit associated to this node.

```
class StiKalmanTrackNode : public StiTrackNode
{
public:
  const StiKalmanTrackNode& operator=(const StiKalmanTrackNode&node);


  double mcs2(double relRadThickness, double beta2, double p2);
  /// Resets the node to a “null” un-used state
  void reset();
  /// Initialize this node with the given hit information
  void initialize(StiHit*h,double alpha, double eta, double curvature,
                double tanl);

  /// Sets the Kalman state of this node equal to that of the given
  /// node.
  void setState(const StiKalmanTrackNode * node);
  // Extract state information from this node.
  void get(double& alpha, double& xRef, double x[5], double cc[15],
          double& chi2);
  /// Get the charge (sign) of the track at this node
  int getCharge() const;
  /// Convenience Method that returns the track momentum at this node
  StThreeVectorF getMomentumF() const;
  /// Convenience Method that returns the track momentum at this node
  /// in global coordinates.
  StThreeVectorF getGlobalMomentumF() const;
  StThreeVector<double> getMomentum() const;
  StThreeVector<double> getGlobalMomentum() const;
  void setDetector(const StiDetector * detector);
  const StiDetector * getDetector() const;
  /// Calculates and returns the momentum and error of the track at this node. The
momentum is
  /// in the local reference frame of this node.
  void getMomentum(double p[3], double e[6]=0) const;
  /// Calculates and returns the tangent of the track pitch angle at this node.
  double getCurvature() const;
  void setCurvature(double curvature);
  double getDipAngle() const;
  double getTanL() const;
```

```
/// Returns the momentum of the track at this node.
double getP() const;
/// Returns the transverse momentum of the track at this node.
double getPt() const;
/// Returns the reference position of this node.
double getRefPosition() const;
/// Returns the reference angle of this node.
double getRefAngle() const;
/// Returns the x,y,z position of this node in global coordinates.
double x_g() const;
double y_g() const;
double z_g() const;

/// Returns the x,y,z position in local detector coordinate of this
/// node.
double getX() const;
double getY() const;
double getZ() const;
/// Return the product of the track curvature and position.
double getEta() const;
/// Returns the chi-square increment associated to the inclusion
/// of this node to the track.
double getChi2() const;
/// modifier
void setChi2(double chi2);
/// Get the local position of this node as a three vector
StThreeVector<double>getPoint() const;
/// Get the global position of this node as a three vector
StThreeVector<double>getGlobalPoint() const;
/// Returns the momentum and error of the track at this node in
/// global coordinates.
void getGlobalMomentum(double p[3], double e[6]=0) const;
```

```
/// Set the attributes of this node as a copy of the given node.
void setAsCopyOf(const StiKalmanTrackNode * node);
/// Propagates a track encapsulated by the given node “p” to the
/// given detector “tDet”.
int  propagate(StiKalmanTrackNode *p, const StiDetector * tDet);
/// Propagates a track encapsulated by the given node “p” to the
/// given vertex
bool propagate(const StiKalmanTrackNode *p, StiHit * vertex);
/// Evaluates, stores and returns the dedx associated with this node.
/// Possible returned values are:
/// > 0 : value of dedx
/// -1  : pathlength was invalid or less than “0”
/// -2  : no hit is associated with the node.
/// -3  : invalid eloss data for this node.
double  evaluateDedx();
int  locate(StiPlacement*place,StiShape*sh);
int  propagate(double x,int option);
void propagateError();
void propagateMCS(StiKalmanTrackNode * previousNode, const
                StiDetector * tDet);
/// Extrapolate the track parameters to radial position “x”  and
/// return a point global coordinates along the track at that point.
StThreeVector<double> getPointAt(double xk) const;

/// propagate the track to the ajusted hit position.
void nudge();
/// evaluate the chi2 increment implied by the addition of the given
/// point to the track.
double evaluateChi2(const StiHit *hit);
/// update the Kalman state of the track
void updateNode();
/// rotate the track internal representation (Kalman state) by
/// the given angle.
void rotate(double alpha);
/// Add the given node as a child of this node.
void add(StiKalmanTrackNode * newChild);
 /// Get the magnetic field
 double getField()  const;
 /// Get the helicity of the track at this node.
 int    getHelicity()  const;
 /// Get the phase of the track at this node.
 double getPhase()   const;
 /// Get a search window size along local y based on the track
 /// parameters and hit error at this node.
 double getWindowY();
 /// Get a search window size along local z based on the track
 /// parameters and hit error at this node.
 double getWindowZ() const;
 /// Return the track dip angle at this node.
 double pitchAngle() const;
 /// Return the track pad row crossing angle at this node.
 double crossAngle() const;
 /// Return the sine of the track pad row crossing angle at this node.
 double sinCrossAngle() const;
 double pathlength() const;
 /// Return the path length between this node and the given node.
 double pathLToNode(const StiKalmanTrackNode * const oNode);

 double length(const StThreeVector<double>& delta, double curv);
 /// Return the track energy loss at this node.
 double getDedx() const;
 /// A helper function to set the given in the range [-pi,pi[
 double nice(double angle) const;
 /// Return center of helix circle in global coordinates
 StThreeVector<double> getHelixCenter() const;

 /// Set track parameters used by the tracker.
 static void   setParameters(StiKalmanTrackFinderParameters *);

 /// Convenience method returns the radiation length of the
 /// material at this node.
 double getX0() const;
 /// Convenience method returns the radiation length through of the
 /// surrounding gas maetrial at this node.
 double getGasX0() const;
 /// Get the density of the material at this node.
 double getDensity() const;
 /// Get the density of the surrounding gas at this node.
 double getGasDensity() const;

 /// Convenience track parameters at this node.
 double _alpha;
 double _cosAlpha;
 double _sinAlpha;
 /// sine and cosine of cross angle
 double _sinCA;
```

```
  double _cosCA;
  /// local X-coordinate of this track (reference plane)
  double _refX;
  double _x;
  /// local Y-coordinate of this track (reference plane)
  double _p0;
  /// local Z-coordinate of this track (reference plane)
  double _p1;
  /// (signed curvature)*(local X-coordinate of helix axis)
  double _p2;
  /// signed curvature [sign = sign(-qB)]
  double _p3;
  /// tangent of the track momentum dip angle
  double _p4;
  /// covariance matrix of the track parameters
  double _c00;
  double _c10, _c11;
  double _c20, _c21, _c22;
  double _c30, _c31, _c32, _c33;
  double _c40, _c41, _c42, _c43, _c44;
  double _chi2;
  float  eyy,ezz;
  short int hitCount;
  short int nullCount;
  short int contiguousHitCount;
  short int contiguousNullCount;
  static StiKalmanTrackFinderParameters * pars;

 protected:
  /// Energy loss calculator
  static const StiElossCalculator * _elossCalculator;
  /// Detector location of this node
  const StiDetector * _detector;

  // some material omitted.
};
```

Figure 7-5 Definition of the StiKalmanTrackNode class.

7.2 Track Factories

The ITTF code uses factories for the instantiation of small objects such as tracks rather than explicit calls to the C++ new operator. Factories are described in Section 5.2. The tracker uses StiKalmanTrack or StiRootDrawableKalmanTrack tracks depending whether the tracker is used simply for analysis production or in the context of the STI event display. In either cases, tracks can be obtained from a factory of type Factory<StiTrack>. Such a factory is defined by default in the ITTF tracker and can be obtained from the Sti toolkit. StiTrack instances may then be recursively obtained by repeatedly calling the getInstance method of the factory as shown in the code example presented in Figure 6-2 below.

```
Factory<StiTrack> * trackFactory = StiToolkit::instance()->getTrackFactory();
StiTrack * track;
For (int I=0;I<1000;I++)
{
  track = trackFactory->getInstance();
// now do something with the track…
}
```

Figure 7-6 How to use the track factory to obtain StiKalmanTrack instances.

In this example, the construct StiToolkit::instance() is used to obtain a pointer to the toolkit singleton. The pointer is then used with the getTrackFactory() method to obtain the actual StiKalmanTrack factory. The factory is then polled 1000 times to obtain 1000 different track instances. The user code must NOT call the delete operator and there shall be no memory leak because the memory management is taken care of by the factory.

The Sti toolkit also provides a StiMcTrack factory (of type Factory<StiMcFactory>) . Tracks of type StiMcTrack are used in the context of the event display for tracker debugging, and tuning.

7.3 Track Filter

A default track filter class, StiDefaultTrackFilter, is provided and used throughout the tracker code and the event display. The class inherits from the abstract templated class EditableFilter<StiTrack>.

7.4 Track Container

The ITTF tracker produces StiKalmanTrack tracks. It also uses, in the context of the event display, and for evaluation purposes, StiMcTrack tracks. Both classes inherit from the common StiTrack base class. We thus implemented a class named StiTrackContainer for the storage and manipulation of these tracks. The StiTrackContainer class definition is shown in Figure 7-7.

The struct StiTrackLessThan defines a functor used to sort tracks on the basis of their transverse momentum. Tracks stored in StiTrackContainers can thus be sorted on the basis of their transverse momentum. Note that one could trivially expand this functionality to include other more sophisticated sorting of the tracks. This could be useful for instance in the context of a V0 calculator or an HBT analysis…

The class StiTrackContainer inherits from the Named and Described classes. Instances of the class can thus be endowed with a name and a description. This is useful in the context of the event display wehre two track containers are used: one for reconstructed tracks and one for monte carlo tracks.

The class StiTrackContainer also inherits from the STL templated map class with template arguments StiTrack* and StiTrackLessThan. Typedef statements are used to define convenience names TrackToTrackMap, and TrackToTrackMapValType as shown in Figure 7-7 below. The StiTrackContainer thus essentially consists of an STL map of tracks to tracks. Note that a map rather than an STL vector is used because it provides for convenient sorting and retrieving capabilities.

The class constructor requires both a name and a description be provided as a way to emphasize that multiple instances of such a container may co-exist.

The methods *void add(StiTrack * track)* and *void push_back(StiTrack*)* provides for two equivalent ways to add tracks to the container.

```
struct StiTrackLessThan
{
    bool operator()(const StiTrack* lhs, const StiTrack* rhs) const;
};

typedef map<StiTrack*, StiTrack*, StiTrackLessThan> TrackToTrackMap;
typedef TrackToTrackMap::value_type TrackToTrackMapValType;

class StiTrackContainer : public TrackToTrackMap, public Named, public
Described
{
public:

    /// Add given track to the container
    void add(StiTrack * track);

    ///Preserve simple interface to add tracks
    void push_back(StiTrack*);

    StiTrackContainer(const string & name, const string & description);
    virtual ~StiTrackContainer();
    int getTrackCount(Filter<StiTrack> * filter) const;

};

inline void StiTrackContainer::add(StiTrack * track)
{
  insert(  TrackToTrackMapValType(track, track) );
}

inline void StiTrackContainer::push_back(StiTrack* track)
{
    insert(  TrackToTrackMapValType(track, track) );
}
```

Figure 7-7 Definition of the StiTrackContainer class.

The *size()* method of the map base class may be called to obtain the number of tracks stored in the container at any one time. We also implemented the *int getTrackCount(Filter<StiTrack> * filter) const* method to return the number of tracks stored in the container that satisfy the given track filter. Note that if no filter is supplied as an argument to getTrackCount, it defaults to returning the size of the map,i.e. the total number of tracks in the container.

One can iterate through all tracks of the container by using a TrackToTrackMap::iterator or TrackToTrackMap::const_iterator as illustrated in Figure

7-8. Note in this example that a pointer to the track container is retrieved from the StiToolkit singleton. The pointer to the container is then used to initialize the iterator *iter* to the beginning (first track) of the container, and to check for the end of the container. In this example, one instantiate a track filter StiDefaultTrackFilter. One then loop over all tracks and apply the filter to extract and print the reduced chi-square of tracks selected the filter.

The StiToolkit maintains two StiTrackContainer containers. They can be retrieved by calling the *StiToolkit ::getTrackContainer()* and *StiToolkit ::getMcTrackContainer()*. These return, as their name suggest, containers used for reconstructed tracks and for monte carlo tracks respectively.

```
StiTrackContainer * trackContainer=
          StiToolkit::instance()->getTrackContainer();
TrackToTrackMap::const_iterator iter;
Filter<StiTrack> * filter = new StiDefaultTrackFilter();
if (!filter) exit(1);
filter->reset();

for (iter=trackContainer->begin();
    iter!=trackContainer->end();
    ++iter)
 {
          if (filter->filter((*iter).second) )
             {
              double chi2 = (*iter).second->getChi2();
              double ndf  = (*iter).second->getPointCount();
              cout << *((*iter).second) << “chi2/ndf:”<< chi2/ndf<<endl;
             }
}
```

Figure 7-8 Example of Iteration through all tracks of an StiTrackContainer.

8 Detector Model Implementation

8.1 Introduction

The ITTF design requirements specify an interactive, three dimensional model of the STAR detector that was faster and easier to control than GEANT, but accurate enough to properly account for corrections for physical processes such as multiple scattering and energy loss, be developed. We explored and developed several different designs. In the end, the choice was made to separate the model of physical objects (air, kapton, silicon, etc) from the hit information that detectors yield. Thus we arrived at two separate groups of classes pertaining to hit handling and detector representation. The detector model implementation is presented in detail in this chapter. The hit representation and organization is discussed in the next chapter.

The software developed to represent, build and use the Sti detector representation involves the following components:

- Representation of detector Modules or elements (e.g. TPC pad row or SVT wafers)
- Geometrical relationship and storage of the detector modules
- Detector traversal (while tracking)
- Detector Groups (e.g. TPC, SVT, etc)
- Detector Builders
- These components and the C++ classes used to represent them are described in the following sections of this chapter.

8.2 Representation of detector elements

The representation of elementary detectors and other structures proceeds with a single class named StiDetector. An instance of the StiDetector class is used to represent a contiguous volume of measuring (or active) devices as well as passive elements. Examples of active devices include a TPC par row, an SVT ladder of Si wafers. Examples of passive elements include beam pipe sections, SVT hybrids.

The representation of a physical structure (passive or active) involves the following aspects and attributes:

- Shape and Size
- Placement (Position and Orientation)
- Whether it is active (a hit measuring device) or passive
- Whether it is contains a gaseous material and can be considered continous
- Whether it is contains a solid material acting as a discrete scatterer
- The material composing the bulk of this object
- The gaseous material, if any, surrounding the object.

The description of the shape and size of the detector structure is accomplished with shape classes. These are discussed in Section 8.2.1. The description of the element placement in the overall detector structure is accomplished with the StiPlacement class. The StiPlacement class is presented in Section 8.2.2. The representation of material properties is accomplished with the StiMaterial class described in Section 8.2.3. The StiDetector class is used to gather all the above information in a single unit. The class is described in Section 8.2.4.

The detector elements (whether active or passive) are all positioned relative to a single mother volume, and their positional relationships are restricted to simple cases. As a specific example, consider that the TPC can be simply represented as a set of 12 identical sectors each consisting of 45 elements. Each TPC pad row is treated as a single volume or unit and no further granularity is assumed. The 540 pad rows forming the STAR TPC are treated on equal footing. One does not use the notion of mother volume, and daughter volumes. Essentially all 540 pad rows are treated equally and no volume containment hierarchy is assumed or used. Instead, we use a simple positional relationship based on a three dimensional sort of the element positions longitudinally, radially and azimuthally. Such a simplified representation is workable for detector geometries such as that of the STAR, PHENIX, or ALICE experiments. A single class, StiDetectorContainer, described in Section , handles the positional relationships, and the storage of the various detector elements.

8.2.1 Shape Representation

The shape portfolio of the tracker is restricted to planar, cylindrical, and conical geometries. TPC pad rows are, for example, appropriately described with a planar geometry, whereas the beam pipe can be easily and simply modeled with a set of cylindrical sections. Three classes are currently defined to represent concrete shapes: StiPlanarShape, StiCylindricalShape, and StiConicalShape. All three are concrete classes inheriting for a common base (and abstract) class named StiShape.

These four classes are rather simple. We therefore limit our discussion to their definition and skip details of implementation.

8.2.1.1 CLASS STISHAPE

This class is an abstract class used to define the essential properties and behavior of shapes in the context of the Sti tracker. This base class encapsulates dimension attributes halfDepth and thickness. halfDepth is the half length of the shape along the beam axis. Thickness is the size of the module in the radial direction. Accessors (get) and modifiers (set), provided for both attributes, are defined in the class header file reproduced in Figure 8-1.

First note that the StiShape inherits from the Named class: it is thus possible albeit optional to name shapes using the detailed class constructor or using the setName() method of the Named class.

Note also that the header file also defines a convenient shape code enumeration list: StiShapeCode which is used to tag the detector element shapes. The accessor method “getShapeCode() const” is implemented in the derived classes.

```
#include “Sti/Base/Named.h”

// allowed values for shapeCode
enum StiShapeCode {kPlanar = 1, kCylindrical, kConical};

/*! Class encapsulating the notion of detector/volume shape.
```

```
\author Ben Norman, Kent State, 25 July 01
\author Claude Pruneau, Wayne State, Oct 2002
*/
class StiShape : public Named
{
public:
   // constructor (for subclass use)
   StiShape(): Named("undefined"),_halfDepth(0.), _thickness(0.){}
   StiShape(const string &name,float halfDepth, float thickness);

   // accessors
   float getHalfDepth() const { return _halfDepth; }
   float getThickness() const { return _thickness; }
   virtual StiShapeCode getShapeCode() const = 0;
   float getEdgeWidth() const { return _edgeWidth; }

   // mutators
   void setHalfDepth(float val);
   void setThickness(float val);

protected:

   double nice(double val);

   /// half extent along z, always >= 0
   float _halfDepth;
   /// "thickness", always >= 0
   float _thickness;
   /// size of the edge used in tracking
   float _edgeWidth;
};
```

Figure 8-1 Header file of the StiShape class.

8.2.1.2 CLASS STIPLANARSHAPE

This class is a concrete class used to represent "rectangular" or box objects. It encapsulates the width, height and depth of the box.

Note that the spirit of the Sti tracker is to use a coarse detector representation. TPC pad rows, which strictly speaking are trapezoidal in shape, are thus instead represented as a flat box with straight edges.

The class is derived from the StiShape class and as such inherits the depth and thickness attributes. It adds the width attributes, and provides both an accessor and a modifiers for this attribute.

The class implements the base class virtual "getShapeCode() const" to return the kPlanar enumeration value.

Instances of this class can be named at construction, or via the setName method of the Named class.

The class header file is listed in Figure 8-2.

```
class StiPlanarShape: public StiShape
{
 public:
  // constructor
  StiPlanarShape(): StiShape(), _halfWidth(0.){}
  StiPlanarShape(const string &name, float halfDepth, float thickness, float
halfWidth);
  // accessors
  float getHalfWidth() const { return _halfWidth; };
  StiShapeCode getShapeCode() const { return kPlanar; };
  // mutators
  void setHalfWidth(float val);
protected:
  float _halfWidth;
};
```

Figure 8-2 Header file of the StiPlanarShape class.

8.2.1.3 CLASS STICYLINDRICALSHAPE

This class is a concrete class used to represent "cylindrical section" or pipe-like objects. It encapsulates the length, radius and thickness of the pipe. Given tracking is done in the local reference frame of the detectors, it is convenient to avoid large rotation by defining cylindrically symmetric objects to be made of multiple cylindrical sections. The beam pipe is for instance represented by 12 cylindrical sections of equal radius.

The class is derived from the StiShape class and as such inherits the depth and thickness attributes of that class. It adds the outerRadius and openingAngle attributes, and provides both accessors and modifiers for these attributes.

The class implements the base class virtual "getShapeCode() const" to return the kCylindrical enumeration value.

Instances of this class can be named at construction, or via the setName method of the Named class.

The class header file is listed in Figure 8-3.

```
#include “StiShape.h”

/*!
  Class to represent a shape within the STAR geometry
  \author Ben Norman, Kent State, 25 July 01
*/
class StiCylindricalShape: public StiShape{
public:

  // constructor
    StiCylindricalShape(): StiShape(), _outerRadius(0.), _openingAngle(0.){}
    StiCylindricalShape(const string &name,
                        float halfDepth_,
                        float thickness_,
                float outerRadius_,
                        float openingAngle_);
    // accessors
    float getOuterRadius() const { return _outerRadius; }
    float getOpeningAngle() const { return _openingAngle; }
    StiShapeCode getShapeCode() const { return kCylindrical; };

    // mutators
    void setOuterRadius(float val);
    void setOpeningAngle(float val);

protected:

    float _outerRadius;  // >= 0
    float _openingAngle; // azimuthal extent of cylinder in radians in [0, 2pi]

};
```

Figure 8-3 Header file of the StiCylindricaShape class.

8.2.2 Detector Placement

The placement of detector elements is accomplished relative to a single global reference frame. Detector elements may be rotated azimuthally only. Although such placement definition is obviously rather restrictive, it can nonetheless accommodate a large of detector features, and is found largely sufficient in the case of STAR.

The placement of detector element is represented with the class StiPlacement whose header file is reproduced in Figure 8-4.

The definition of this class is predicated by a desire for speed. A certain level of information redundancy is thus used.

The class defines the following attributes: normalRefAngle, normalRadius, normalYoffset, centerRefAngle, centerRadius, centerOrientation [-pi/2, pi/2)

LayerRadius, and zCenter. Accessors and modifiers of these attributes are also provided.

The “center” of a planar detector is its center of gravity (the mindpoint in local x, y, and z). For curved (cylindrical or conical sections) detectors, the “center” is the midpoint in z, opening angle, and radial thickness.

The “normal” coordinates give the magnitude and azimuthal angle of a normal vector from the global origin to the plane of the detector. The plane of a planar detector is simply the one in which it lies. For a curved detector, the plane is that which contains the tangent to the curved section at its center and is normal to the transverse projection of the radial vector from the origin to its center. Note that these definitions all assume that the plane of the detector is parallel to global z. The third “normal”

cooridnate gives the location of the detector center along the detector plane in the azimuthal direction (i.e., local y). This representation is best for the Kalman local track model.

The “center” coordinates are a little more natural and are best used for rendering and radial ordering. Here, the magnitude and azimuthal angle of a vector from the global origin to the center of the detector are given, as well as an orientation angle. The orientation angle is the angle from the vector above to the detector plane’s outward normal. It is 0 for detectors which have xOffset==0. When setting the values, one must set all 3 for a representation at once. The other representation is then recalculated and both are available for quick access.

The layerRadius is independent and is used for ordering detectors in R by the detector container.

```
class StiPlacement
{
public:
  enum StiRegion {kBackward, kMid, kForward, kUndefined};
  StiPlacement();

    // accessors
    float getNormalRefAngle() const { return normalRefAngle; }
    float getNormalRadius() const { return normalRadius; }
    float getNormalYoffset() const { return normalYoffset; }
    float getCenterRefAngle() const { return centerRefAngle; }
    float getCenterRadius() const { return centerRadius; }
    float getCenterOrientation() const { return centerOrientation; }
    float getLayerRadius() const { return layerRadius; }
    float getZcenter() const { return zCenter; }

    // mutators
    void setNormalRep(float refAngle_, float radius_, float xOffset_);
    void setCenterRep(float refAngle_, float radius_, float orientation_);
    void setLayerRadius(float radius_){ if(radius_>=0) layerRadius = radius_;
}
    void setZcenter(float val){ zCenter = val; }

protected:

    // store both representations
    float normalRefAngle; // in [-pi, pi)
    float normalRadius;   // >= 0
    float normalYoffset;
    float centerRefAngle; // in [-pi, pi)
    float centerRadius;   // >= 0
    float centerOrientation;  // in [-pi/2, pi/2)

    // independent radius for ordering
    float layerRadius;
    float zCenter;

};
```

Figure 8-4 Header file of the StiPlacement class.

8.2.3 Material Representation

For track reconstruction purposes, one requires effective values for the density, the radiation length, and the ionization potential of the materials. It is also convenient to associate a name and store the atomic and mass numbers of each relevant material.

Material representation is accomplished with instances of the StiMaterial class whose header file is reproduced in Figure 8-5.

The class encapsulates the attributes “z” (effective atomic number), “a” (effective mass number), “density” (in g/cm^3), “radLength” (radiation length in g/cm^2), “ionization” potential (effective ionization in eV), “zOverA” ratio, and “x0” (radiation length in cm). Accessors, and modifiers are also provided for all these attributes.

Instances of this class can be named at construction, or via the setName method of the Named class.

The class header file is listed in Figure 8-3.

```
class StiMaterial : public Named
{
public:
   StiMaterial();
   StiMaterial(const string &name,double z,double a,double density,double
   radLength,double ionization);
   virtual ~StiMaterial();
   /// Get the material density in grams/cubic centimeter
   double getDensity() const { return _density; }
   /// Get the radiation length in g/cm^2
   double getRadLength() const { return _radLength; }
   /// Get the radiation length in centimeter
   double getX0() const { return _x0; }
   /// Get the effective atomic mass of the material
   double getA() const { return _a;}
   /// Get the effective atomic number of the material
   double getZ() const { return _z;}
   /// Get the effectice ionization potential of the material
   double getIonization() const { return _ionization;}
   /// Get Z over A ratio
   double getZOverA() { return _zOverA;}
   void set(const string& name,
            double a,
            double z,
            double density,
            double radLength,
            double ionization);
protected:
   /// Effective Z
   double _z;
   /// Effective A
   double _a;
   /// g/cm^3
   double _density;
   /// radiation length in g/cm^2
   double _radLength;
   /// Effective ionization (in eV)
   double _ionization;
   /// zOverA
   double _zOverA;
   /// radiation length in cm.
   double _x0;
};
```

Figure 8-5 Header file of the StiMaterial class.

8.2.4 Detector Element Representation

The overall modeling and representation of detector elements is carried by the StiDetector class. This inherit from the Named class so detector elements (or instance) can be given a name. The class essentially acts as a container of all the properties and attributes of detector elements. As such, it features accessor and modifiers methods to get and set the many different attributes of a detector element. It also provides a few action methods discussed in the following.

The StiDetector class definition is presented in Figure 8-6. The data member are documented within the class definition and their interpretation is straightforward.

```
class StiMaterial;
class StiShape;
class StiIsActiveFunctor;
template<class T> class StiCompositeTreeNode;
class StiHitErrorCalculator;

class StiDetector : public Named
{
public:
  StiDetector();
   virtual ~StiDetector();
   bool isOn() const {return on;}
   inline bool isActive(double dYlocal, double dZlocal) const;
   inline bool isActive() const;
   bool isContinuousMedium() const { return continuousMedium; }
   bool isDiscreteScatterer() const { return discreteScatterer; }
   StiMaterial* getGas() const { return gas; }
   StiMaterial* getMaterial() const { return material; }
   StiShape* getShape() const { return shape; }
   StiPlacement* getPlacement() const { return placement; }
   void setIsOn(bool val) {on = val;}
   void setIsActive(StiIsActiveFunctor *val){ isActiveFunctor = val; }
   void setIsContinuousMedium(bool val) {continuousMedium = val;}
   void setIsDiscreteScatterer(bool val) {discreteScatterer = val;}
   void setGas(StiMaterial *val){ gas = val; }
   void setMaterial(StiMaterial *val){ material = val; }
   void setShape(StiShape *val){ shape = val; }
   void setPlacement(StiPlacement *val);
   virtual void build(){}  //for now, build from SCL parsable ascii file
   void setTreeNode( StiCompositeTreeNode<StiDetector> * val)
{mNode=val;}
   StiCompositeTreeNode<StiDetector> * getTreeNode() const {return
mNode;}

   void setHitErrorCalculator(const StiHitErrorCalculator * calculator);
   const StiHitErrorCalculator * getHitErrorCalculator() const;
protected:
   /// Toggle switch determining whether this detector is to be added to the
detector tree.
   /// The detector is added if the switch is "true"
   bool on;
   /// Functor used to calculate whether the posistion reached by a track is
   /// to be considered within the active area of the detector, and
   /// is thus susceptible of providing hit information.
   StiIsActiveFunctor *isActiveFunctor;
   /// Toggle switch determining whether this detector contains a continuous
   /// medium (e.g. gas). If true, scatterer information is provided
   /// by the gas material.
   bool continuousMedium;
   /// Toggle switch determining whether the detector contains a discrete
   /// thin scatterer (e.g. a Si wafer). If true, scatter information provided
   /// by the material.
   bool discreteScatterer;   // is this a discrete scatterer?    (yes => scatterer
given by "material" below)
   /// Hit Error Calculator for this detector
   const StiHitErrorCalculator * _hitErrorCalculator;
   /// Continuous scatter attributes.
   StiMaterial *gas;
   /// Discrete scatterer attributes
   StiMaterial *material;
   /// Physical Shape attribute of this detector or voloume
   StiShape    *shape;
   /// Physical position and orientation of this detector or volume.
   StiPlacement *placement;
   /// Pointer to the parent detector node.
   StiCompositeTreeNode<StiDetector>  * mNode;
   /// Convenience storage of cos(refAngle)
   double _cos;
   /// Convenience storage of sin(refAngle)
   double _sin;
};
```

Figure 8-6 Definition of the StiDetector class.

The “on” Boolean flag determines whether the detector element is in use. It should allow in principle to dynamically remove or add detector elements. In practice, in the current version of the code, this flag is not used.

The “isActiveFunctor” is, as its name suggests, a pointer to a StiIsActiveFunctor. The functors is called within the “isActive()” method to determine whether a given point (expressed in local detector coordinates) is within the active area of the detector. StiIsActiveFunctors when instantiated query the STAR database to determine whether the given detector module was active during the analysis of the data run of interest.

The Boolean flags continuousMedium and discreteScatterer are used to dictate the behavior of the detector element.

The variable _hitErrorCalculator is a pointer to a hit error calculator appropriate for the detector element. It must be initialized after the object is instantiated. This is typically done by a detector builder.

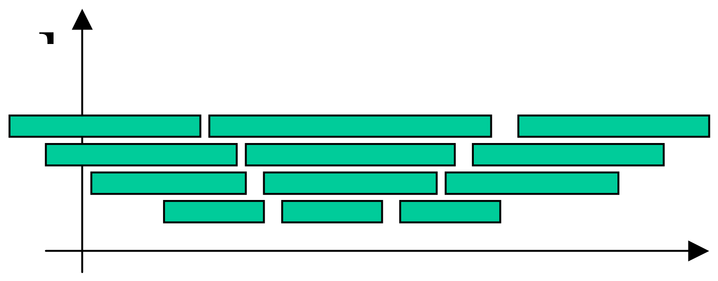
The variables material and gas, are pointers to StiMaterial objects describing respectively the main material of the detector element, and its surrounding gas if any.

The variables shape and placement are pointers to StiShape and StiPlacement objects describing respectively the shape and position of the detector module.

mNode is a pointer to StiCompositeTreeNode<StiDetector> node used by the StiDectectorContainer for detector traversal while tracking.

8.3 Detector Model – The big picture

The detector elements are represented, as discussed in the previous section, by instances of the StiDetctor class. The StiDetector class provides, through its StiPlacement member, information about the position and orientation of the detector. It however does not provide for a structure to assemble and store all the detector modules. This is accomplished by a class named StiDetectorContainer. We stress once again that unlike the Mother-Daughter volume hierarchical model used by GEANT, the Sti code uses a simple (democratic) flat model where detector positioning is accomplished in relationship to neighbors rather than by inclusion into a parent. The neighbor relationship is predicated on a three dimensional sort of the position of the detector. It assumes that the volumes are well behaved (in the context of the model) and do not overlap.



Representing the scattering materials is not particularly challenging. The real challenge lies in organizing them in such a way that navigation through the detector model is robust, fast, and flexible. After many different attempts we settled on the choice of an ordered tree structure. Thus, we separate the detector using the following hierarchy: regions, then radius, then azimuth. This separation takes advantage of the symmetry of Collider detectors and allows for a unique sorting of the detectors, which further allows for *extremely* efficient navigation through the detector model. Further, the ordering is in essence a generalized cylindrical coordinate system in which one can use a unique (and extremely efficient) helical track model.

We first recognize that Collider detectors have two regions of tracking: mid-rapidity and forward/backward rapidity. Specifically, we have SVT, TPC, TOF, RICH, and EMC in the mid-rapidity region. In the forward region we have FTPC, BBC, bEMC, and FPD, for now. Track finding and fitting generally happens via progression from outside-to-in (or backwards) in one or the other of these two regions.

Next, for each region we order "layers" by radius. That is, the radial distance from the origin. This reflects the fact that most detectors can be "peeled" like an onion. This allows for a simple track propagation from one layer to the next. While we use this fact for optimization, we do allow for the occurrence of more than one scattering center at the same radial position (and azimuthal position), so we do not oversimplify the problem with the radial ordering. Finally, we order the elements within a given radial layer by azimuth.

The detector model is implemented by class StiDetectorContainer. StiDetectorContainer essentially behaves as an ordered tree of detector elements of type StiDetector. The tree structure is implemented using class the templated class StiCompositeTreeNode<T>. Thus, each node has a single parent and a variable number of children that are stored in a sorted container of type std::vector<StiCompositeTreeNode*>.

The tree structure is built dynamically at run-time by the pure abstract class StiDetectorBuilder which is implemented in a derived class StiCodedDetectorBuilder. StiCodedDetectorBuilder queries the offline database for information and builds each detector element with that information, getting the actual StiDetectorObjects from a factory. The two-level design was chosen to provide a natural place for building the detector model from information from other sources, e.g., a geant file. To do so, one simply derives a new class from StiDetectorBuilder and implements the StiDetector* getNextDetector() method.

Navigation of the tree makes use of the aforementioned sorting. The StiDetectorContainer makes use of several iterators, one iterator for each classification of sorting. Therefore there is an iterator into the region (forward/backward or mid-rapidity), into the layer (radial position), and into the azimuth. These iterators are of type StiCompositeTreeNodeIterator, which is part of the StiCompositeTreeNode package.

These iterators fulfill the ANSI requirements of a bi-directional iterator, and can thus be used for any of the STL algorithms. Navigation consists of setting the detector model to a starting point, specified by either a region, position, and azimuth, or by a StiDetector object. Next, one moves in or out using the methods moveIn() or moveOut(). When either of these methods are called, the detector container chooses the element in the next layer that is closest in azimuth to the layer that we are propagating from. Then one dereferences the StiDetectorContainer to get a pointer to the current detector element. One can also move in azimuth using the functions movePlusPhi() and moveMinusPhi().

The main time saving optimization lies within these methods. Within a given layer, the elements are sorted in order of increasing azimuth. Thus, finding the element closest to a given azimuth is quickly performed using the STL binary search algorithm. If a layer has less then a given number (~15) elements in azimuth, the container actually uses a linear search algorithm instead. However, there is an even stronger optimization.

Because many layers of the detector are so symmetric (e.g., layers within a tpc), a call to moveIn() or moveOut() doesn't necessarily have to do a search to find the element closest in azimuth. Instead, the detector container can check whether the new layer and the old layer have the same symmetry (number of azimuthal divisions, angular offset, etc). If this condition is true, then the detector container simply indexes into the container of detector elements. If a layer has 'N" elements, then a linear search takes time O(N), a binary_search takes time O(log(N)), but indexing into the container takes constant time, which is much faster than any type of search. Because the majority of the time is spent swimming in through 45 layers of the tpc, this represents a tremendous savings.

There is no way that we can reflect the full implementation of the detector model in this document. However, we highlight the most important lessons that we learned in the iterative design process.

A collider detector is well mapped to generalized cylindrical coordinates

A 3-level sorting (which can be easily extended) accounts well for representation and navigation through all of the components in (and designed for the future) STAR detector.

Strict use of STL containers, iterators, and algorithms shave tens of cpu seconds per event off of the reconstruction time.

A *sorted* composite tree structure is perhaps the best balance of efficiency and extendibility. This choice was made after testing many different ideas including the following: a many dimensional lattice, a STL mutlimap, and a polygonal geometry.

9 New Detector Implementation

9.1 Introduction

The tracker is designed to enable easy integration of new detector components. Example of components of interest are the STAR FTPC, EMC, and a possible new PIXEL detector. This section presents a detail description of the steps to be followed to incorporate a new detector in the STAR tracker. We preface this discussion by stating that we use the notion of detector group to encapsulates the detectors with common functionality. For instances all pad rows of the TPC are collectively known as the TPCgroup; the 6 layers of the SVT are collectively known as the SVT group, etc. The addition of a new detector thus proceeds via the addition of a new detector group. The implementation of a detector group named "Xxx" requires the following software elements:

- Addition of a new software package under StRoot called StiXxx.
- Concrete classes implementations:
  - Class StiXxxDetectorGroup
  - Class StiXxxDetectorBuilder
  - Class StiXxxHitLoader
  - Class StiXxxIsActiveFunctor
  - Class StiXxxHitErrorCalculator (optional)
- Modifications to StiMaker/macros/Run.C and StiMaker/StiMaker.cxx files to explicitely include the new detector groups.
- Package additions are handled by the STAR librairian and should be preapproved by the computing leader. Implementation of the above classes is best done by a member of the development team of the new detector whereas modifications to the Run macro and the StiMaker should be done by a member of the tracking group. Example implementation of all five classes to be developed by the detector team are provided in the following sections.

9.2 StiXxx Concrete Class Implementation

The various class implementations required for the addition of new detector groups are provided in the following subsections.

9.2.1 Class StiXxxDetectorGroup

The StiXxxDetectorGroup class is a convenience class used to regrouped all functionalities needed for that detector. These essentially include pointers to a builder and a hit loader. The group class is also used to instantiate and set the appropriate isActiveFunctor and HitErrorCalculator.

As shown in the code snipet displayed in Figure 9-1, StiXxxDetectorGroup is a concrete class which inherits from the abstract (and templated) StiDetectorGroup class. The construction (instantiation) of StiXxxDetectorGroup is performed once by StiMaker.

```
class StiXxxDetectorGroup : public StiXxxGroup<StEvent,StMcEvent>
{
 public:
   StiPixelDetectorGroup(bool active);
   ~StiPixelDetectorGroup();
};
```
Figure 9-1 Example of a header file for the StiXxxDetectorGroup class.

A typical implementation of the constructor is shown in Figure 9-2. The constructor takes one argument "active" which indicates whether the detector is considered as a measuring device, i.e. bearing hits, or as passive volume and scattering center. The constructor of the StiDetctorGroup base class in turn takes three arguments: the name of the detector group, a hit loader instance, and a detector builder appropriate for this group. These are described in following sections. Note that the active parameter is used to decide whether a hit loader should be instantiated, and is passed to the builder so each detector module be set with the proper value of their active flag.

```
StiXxxDetectorGroup::StiXxxDetectorGroup(bool active)
   : StiDetectorGroup<StEvent,StMcEvent>("Xxx",
                                          active?new StiXxxHitLoader():0,
                                          new StiXxxDetectorBuilder(active),0,0)
{
 /* no operation */
}
```
Figure 9-2 Example of implementation of the constructor of the StiXxxDetectorGroup class.

9.2.2 Class StiXxxDetectorBuilder

The StiXxxDetectorBuilder is a concrete class derived from the abstract StiDetectorBuilder class. It has the responsibility to build, i.e. to instantiate, the geometry and all components of the Xxx detector. As derived class of StiDetectorBuilder, it must implement the methods listed in Figure 9-3.

```
class StiXxxDetectorBuilder : public StiDetectorBuilder
{
public:
   StiXxxDetectorBuilder(bool active);
   virtual ~StiXxxDetectorBuilder();
   virtual void loadDb();
   virtual void buildMaterials();
   virtual void buildShapes();
   virtual void buildDetectors();
   double phiForXxxSector(unsigned int iSector) const;
};
```
Figure 9-3 Example of a header file for the StiXxxDetectorBuilder class.

An example of implementation of the constructor StiPixelDetectorBuilder(bool active) is presented in Figure 9-4. The argument "active" of the constructor is passed to the constructor of the base class which takes care of storage. A name is also passed to the base class constructor as the first argument. The constructor is used to instantiate and store a pointer to a hit calculator appropriate for this detector group. Here a default calculator, StiDefaultHitErrorCalculator, is used.

```
StiPixelDetectorBuilder::StiPixelDetectorBuilder(bool active)
  : StiDetectorBuilder("PixelBuilder",active)
{
   // Parameterized hit error calculator.
   // Given a track (dip, cross, pt, etc) returns average error
   // once you actually want to do tracking, the results depend
   // strongly on the numbers below.
   _hitCalculator = new StiDefaultHitErrorCalculator();
   _ hitCalculator->set(0.00004, 0., 0., 0.00004, 0., 0.);
}
```

Figure 9-4 Example of a DetectorBuilder constructor.

The next step in the definition of a detector consists in the instantiation of StiMaterail objects encapsulating the properties of the relevant materials. This is done with the "buildMaterials()". An example of the implementation of this method for the pixel detector is shown in Figure 9-5. In this example, two material objects are defined and stored to pointers _gas and _fcMaterial (assumed to be defined in the header file of the class). The two object pointers are also added to internal storage of the builder by a call to the "add()" method.

```
void StiPixelDetectorBuilder::buildMaterials()
{
 //_gas is the gas that the pixel detector lives in
 _gas= add(new StiMaterial("Air",0.49919,1.,0.0012,30420.*0.001205,5.));
 //_fcMaterial is the (average) material that makes up the detector
 // elements.  Here I use ~silicon
 _fcMaterial=add(new StiMaterial("Si",14.,28.0855,2.33,21.82,5.) );
}
```

Figure 9-5 Example implementation of the buildMaterials() method for a pixel detector

The next step in the definition of a detector involves the definition of volume or shapes. This may be done through a call to the "buildShapes()" method. An example of an implementation for the Pixel detector is shown in Figure 9-6. In this example, a cylindrical shape object is instantiated and set. The "setThickness()" is used to set the radial thickness of the cylinder. The "setHalfDepth" method is used to set the length (or depth) along the z direction. The "setOpeningAngle()" is used to set the size of the angular span covered by this volume. The "setOuterRadius()" method defines the outer radius of the shape. The volume shape is finally added to the builder storage with the "add" method.

```
void StiPixelDetectorBuilder::buildShapes()
{
  double pixRadius = 5.0; //cm
  StiCylindricalShape *ifcShape = new StiCylindricalShape;
  ifcShape->setName("Pixel/sector");
  ifcShape->setThickness(0.1); //radial thickness in cm
  ifcShape->setHalfDepth( 16./2. ); //half depth along beam direction
  //the angle swept out.  So, for 12 segments, = pi/6
  ifcShape->setOpeningAngle( M_PI/6. );
  ifcShape->setOuterRadius(pixRadius + ifcShape->getThickness()/2.);
  add(ifcShape);
}
```

Figure 9-6 Example implementation of the buildShapes() method for a pixel detector.

The final step is to put all the information together and actually build the detector objects. This is accomplished within the "buildDetectors()" method.

An example implementation for a Pixel detector is shown in Figure 9-7. First, the required shape is retrieved by name with the findShape("Pixel/sector") call . Note here than an exception is thrown if a null pointer is returned. In this example, the detector is assumed to consist of one row (nRows=1), and 12 sectors. A for loop is used to define all detector elements.

The definition of each element proceeds as follows: (1) a placement object (StiPlacement) is instantiated to represent the position and orientation of the detector element. One first sets the position along the beam axis with the "setZcenter()" method. The radial position is set with the "setLayerRadius()". The setRegion call is used to determine whether this detector is at midrapidity or lies on one of the two end caps. In this example, all detector elements are to be inserted in the mid rapidity region. One therefore sets the placement to StiPlacement::kMidRapidity. Next, one invokes a call to the detector object factory to get a pointer to a detector object. This detector object is then setup. One declare its name; set the volume to be "On"; One next declares the volume to be active and set the speficifc active functor to be used for the determination of active volumes. See the section on ActiveFunctors to understand their role and find implementation examples.

Detector and volumes are composite. They can be discrete solid object or continuous gas volumes. They can also contain a main material as well as a surrounding material. One must then declare, as it is done in this example,whether the volume is continious volume (e.g. TPC gas) or a discrete scatter (e.g. SVT wafer). The detector shape is set with the "setShape" method. The placement is set next with the "setPlacement" call. Next is the main material, and then the surrounding gas with "setGas()". The last set method call associates a specific hit error calculator to the given detector. Finally the detector is added a the specified row and sector indices with a call to the "add" method as shown.

```
void StiPixelDetectorBuilder::buildDetectors()
{
 char name[50];
 StiShape *ifcShape = findShape(“Pixel/sector”);
 if (!ifcShape)
    throw runtime_error(
     “StiPixelDetectorBuilder::buildDetectors() - FATAL - ifcShape==0”);
 double pixRadius = 5.0;
 unsigned int nRows=1;
 StiPlacement *p;
 for (unsigned int row=0; row<nRows; ++row)
   {
    for(unsigned int sector = 0; sector<12; ++sector)
      {
          p = new StiPlacement; //see StRoot/Sti/StiPlacement.h
          p->setZcenter(0.);
          p->setLayerRadius(pixRadius);
          p->setRegion(StiPlacement::kMidRapidity);
          p->setNormalRep(phiForTpcSector(sector), pixRadius, 0.);
          StiDetector *detector = _detectorFactory->getInstance();
          sprintf(name, “Pixel/Layer1/Sector_%d”, sector);
          detector->setName(name);
          detector->setIsOn(true);
          detector->setIsActive(new StiPixelIsActiveFunctor);
      detector->setIsContinuousMedium(false);
          detector->setIsDiscreteScatterer(true);
          detector->setShape(ifcShape);
          detector->setPlacement(p);
          detector->setGas(_gas);
          detector->setMaterial(_fcMaterial);
          //set the pointer to the hit error calculator
          detector->setHitErrorCalculator(_innerCalc);
          add(row, sector, detector); //this hangs it in the right place!
 } } }
```

Figure 9-7 Example implementation of the buildDetectors() method for a pixel detector.

The loadDd() method is provided to allow detector code writers to have a dedicated location for the fetching of db related information. Figure 9-8 provides an example based on the TPC implementa.tion. Here one first gets verify that the global tpc pointer “gStTpcDb” points to a valid object, and issue a runtime exception if it does not. One then proceeds to extract pointers to the padPlane and dimensions. One verify that valid objects are returned and again throw exception if they are not. One next instantiates a TpcCoordinateTransform functor and verify that it is properly instantiated. The geometry is then extracted fomr the db. One gets the pad plane info, the number of pad rows, and the number of sectors for each pad row. The “setNRows()” method is used to setup an internal array that will hold the geometry of this detector. One iterates through all rows and set the number of sectors to 12 for each row. Note that one could set different number of sectors for each row as in the case of the SVT.

```
void StiTpcDetectorBuilder::loadDb()
{
if (!gStTpcDb)
 throw runtime_error("StiTpcDetectorBuilder::buildShapes() -E- gStTpcDb==0");
_padPlane = gStTpcDb->PadPlaneGeometry();
if (!_padPlane)
  throw runtime_error("StiTpcDetectorBuilder::buildShapes() -E-
_padPlane==0");
_dimensions = gStTpcDb->Dimensions();
if (!_dimensions)
  throw runtime_error("StiTpcDetectorBuilder::loadDb() -E- dimensions==0");
_transform = new StTpcCoordinateTransform(gStTpcDb);
if (!_transform)
 throw runtime_error("StiTpcDetectorBuilder::loadDb() –E- gStTpcDb==0");
StTpcPadPlaneI *_padPlane = gStTpcDb->PadPlaneGeometry();
if (!_padPlane)
 throw runtime_error("StiTpcDetectorBuilder::loadDb() –E- _padPlane==0");
unsigned int nRows = _padPlane->numberOfRows();
setNRows(nRows);
for(unsigned int row = 0; row<nRows;row++)
  setNSectors(row,12);
}
```

Figure 9-8 Implementation example of the loadDb() based on the TPC implementation.

9.2.3 Class StiXxxHitLoader

The class StiXxxHitLoader is responsible to load hits of the Xxx detector from an arbitrary source into the appropriate StiHitContainer. As such, it is a concrete class based on the abstract class StiHitLoader. Loaders must be supplied for all detector groups. It is also possible in principle, to have many loaders for a given group. This is appropriate for instance, if one wishes to load hits from different types of sources. Two examples of such sources are Star staff tables and STAR StEvent persistent model. The toolkit should then be the directing mechanism to instantiate and load the appropriate loader at run time.

The loaders are nominally expected to load hits either from StEvent or from StMcEvent. The StiXxxHitLoader must then implement two methods to enable the load of either Monte Carlo or reconstructed hits, or both. The base class also provides an option to enable the load of Monte Carlohits as if they were actual reconstructed hits. That is the "useRecAsMc" flag.

Here we describe the implementation of the class constructor as well as the implementation of the loadHits() and loadMcHits() methods.

Figure 9-9 provides an example of a header file for the definition of the derived class StiTpcHitLoader. The class is defined to inherit from the templated StiHitLoader class with concrete object types StEvent, StMcEvent, and StiDetectorBuilder. One could in principle have derived class using hit sources other than StEvent and StMcEvent by substituting the appropriate types here.

The constructor takes four arguments as follows: (1) pointer to the hit container where reconstructed hits (nominally from StEvent) should be loaded to; (2) pointer to the hit container where the MC hits (if any) should be loaded to; (3) a hit object factory; (4) a detector builder. The first hit container is used for reconstruction purposes whereas the second (MC) is used for evaluation while using the code either in graphics mode or with an internal hit associator for evaluation purposes. The hit factory should be used in lieu of the “new” operator to optimize memory management and guarantee there are no memory leaks created by the user code. Example of its use are provided below. The detector builder is needed to establish a link between the hits and their measuring detector. An example of this association is given also in the following.

The StiXxxHitLoader class must implement the “loadHits” and the “loadMcHits” as shown in Figure 9-9. Implementation examples follow also.

```
Class StiTpcHitLoader : public
StiHitLoader<StEvent,StMcEvent,StiDetectorBuilder>
{
 public:
  StiTpcHitLoader(StiHitContainer * hitContainer,
              StiHitContainer * mcHitContainer,
                      Factory<StiHit> * hitFactory,
                      StiDetectorBuilder * detector);
  virtual ~StiTpcHitLoader();
  virtual void loadHits(StEvent* source,
                          Filter<StiTrack> * trackFilter,
                          Filter<StiHit> * hitFilter);
  virtual void loadMcHits(StMcEvent* source,bool useMcAsRec,
                           Filter<StiTrack> * trackFilter,
                           Filter<StiHit> * hitFilter);
};
```

Figure 9-9 Example of a hit loader definition based on the STAR TPC.

Figure 9-10 presents an example implementation of the StiXxxHitLoader for the SVT detector. Essentially, nothing has to be done in the constructor of the derived class other than forward the parameters of the class to that of the base class. Note however that a name for this loader is given as the first argument passed to the base class.

```
StiSvtHitLoader::StiSvtHitLoader(StiHitContainer* hitContainer,

StiSvtHitLoader::StiSvtHitLoader(StiHitContainer* hitContainer,
                                  StiHitContainer* mcHitContainer,
                                  Factory<StiHit>*hitFactory,
                                  StiDetectorBuilder*detector)
  : StiHitLoader<StEvent,StMcEvent,StiDetectorBuilder>("SvtHitLoader",
hitContainer,
mcHitContainer,
hitFactory,
detector)
{}
```

Figure 9-10 Example of contructor implementation for the StiXxxHitLoader class.

The implementation of the actual loader is where the developer's "work" resides. Example implementations of the hit and mcHit loaders for the STAR TPC are presented in Figure 9-11 and Figure 9-12 respectively. In these two examples, one extracts the hit information from its original format in StEvent and StMcEvent by looping respectively on StTpcHitContainers and StMcTracks. The code itself is rather self-explanatory. Four important points are to be stressed. (1) The external (StEvent or StMcEvent) and internal (Sti) detector representation are slightly different. One must thus effect a translation of the row and sector numbers as highlighted in blue in the example below. (2) The loader does not the operator "new" to instantiate StiHit. Instead it invokes a call to a hit factory (highlighted in red below). (3) The StiHit instance is set using a STAR global reference frame although internally, the StiHit class uses coordinates local to the detector where the hit was measured (in green below). (4) Track and hit filters can in principle be applied to decide whether one should include or exclude hits and tracks. In the examples presented in , the hit loader does not use any particular filter. The mc hit loader shown in however uses a track filter to decide what hits must be copied to internal StiHitContainers.

```
void StiTpcHitLoader::loadHits(StEvent* source,
                               Filter<StiTrack> * trackFilter,
                               Filter<StiHit> * hitFilter)
{
 cout << "StiTpcHitLoader::loadHits(StEvent*) -I- Started" << endl;
 //make sure we have a valid builder, and throw a runtime exception
 //if we don't
 if (!_detector)
   throw runtime_error("StiTpcHitLoader::loadHits() -F- _detector==0");
 if(!_hitContainer)
   throw runtime_error("StiTpcHitLoader::loadHits() -F - _hitContainer==0");
 //define some local symbols
 StiDetector * detector;
 StiHit* stiHit;
 //Get the hit collection for TPC from StEvent
 const StTpcHitCollection* tpcHits = source->tpcHitCollection();
```

```
unsigned int stiSector;
// loop on all TPC sectors
for (unsigned int sector=0; sector<24; sector++)
  {
   //translate STAR sector into StiTpc sectors 1-24 becomes 0-11
   if (sector<12)
    stiSector = sector;
   else
    stiSector = 11 - (sector-11)%12;
   const StTpcSectorHitCollection* secHits = tpcHits->sector(sector);
   if (!secHits) break;
   //loop on all rows
   for (unsigned int row=0; row<45; row++)
        {
         const StTpcPadrowHitCollection* padrowHits = secHits->padrow(row);
     // skip if this row has not hits
         if (!padrowHits) break;
         //extract the hits
     const StSPtrVecTpcHit& hitvec = padrowHits->hits();
         detector = _detector->getDetector(row,stiSector);
         //throw runtime exception if the detector is not defined.
         if (!detector)
            throw runtime_error("StiTpcHitLoader::loadHits(StEvent*) -E-
Detector element not found");
     //loop on hits
         for (vector<StTpcHit*>::const_iterator iter = hitvec.begin();
              iter != hitvec.end();
              iter++)
            {
            StTpcHit*hit=*iter;
            if(!_hitFactory)
             throw runtime_error("StiTpcHitLoader::loadHits() -E-
hitFactory==0");
            stiHit = _hitFactory->getInstance();
            if(!stiHit)
                throw runtime_error("StiTpcHitLoader::loadHits() -E- stiHit==0");
            stiHit->reset();
        //set the attributes of this hit in global coordinates
            stiHit->setGlobal(detector,
                              hit,
                              hit->position().x(),
                              hit->position().y(),
                              hit->position().z(),
                              hit->charge());
            //add this hit to the hitContainer
            _hitContainer->push_back( stiHit );
            }
          }
    }
  }
```

Figure 9-11 Anoted Example implementation of the loadHits(…) method.

The loadMcHits() method operates in a similar way as the “loadHits()” method although it gets hits from tracks rather than detector elements. Here again, hits are obtained from a hit factory, and the StiHit are set using global coordinates. Note that the Monte Carlo and the reconstructed hits are stored in ITTF using instances of the same class StiHit. They are stored in two different instances of the same type of container StiHitContainer.

```
void StiTpcHitLoader::loadMcHits(StMcEvent* source,
                                  bool useMcAsRec,
                                  Filter<StiTrack> * trackFilter,
                                  Filter<StiHit> * hitFilter)
{
  cout << “StiTpcHitLoader::loadMcHits(StEvent*) -I- Started” << endl;
  //Start with sanity checks.
  if (!_detector)
    throw runtime_error(“StiTpcHitLoader::loadMcHits() -F- _detector==0”);
  if(!_mcHitContainer)
    throw runtime_error(“StiTpcHitLoader::loadMcHits() -F-
_mcHitContainer==0”);
  if(!_mcTrackFactory)
    throw runtime_error(“StiTpcHitLoader::loadMcHits() -F-
_mcTrackFactory==0”);
  if (!_mcTrackContainer)
    throw runtime_error(“StiTpcHitLoader::loadMcHitss() -F-
_mcTrackContainer==0”);
  if(!_hitFactory)
    throw runtime_error(“StiTpcHitLoader::loadMcHits() -F- _hitFactory==0”);
  //Extract vector of mc tracks from StMcEvent
  StSPtrVecMcTrack & mcTracks = source->tracks();
  StiMcTrack * mcTrack;
  StMcTrack  * stMcTrack;
  StMcTrackConstIterator iter;
  //loop on all tracks
  for (iter=mcTracks.begin();iter!=mcTracks.end();iter++)
    {
      stMcTrack = *iter;
      //extract the kinematical info
      double eta = stMcTrack->pseudoRapidity();
      double pt  = stMcTrack->pt();
      //extract the hits
```

```
     const StPtrVecMcTpcHit& hits = stMcTrack->tpcHits();
     int nPts = hits.size();
     //get a pointer to an StiTrack instance
     mcTrack = _mcTrackFactory->getInstance();
     //reset this instance, and then copy the kinematic info
     mcTrack->reset();
     mcTrack->setStMcTrack( (*iter) );
     double charge = mcTrack->getCharge();
     //Use a filter to decide whether to save the track into
     //the Sti track container
     if (!trackFilter ||
         trackFilter->filter(mcTrack) )
          {
           _mcTrackContainer->add(mcTrack);
       //loop on the hits of the StMcTrack
           for (vector<StMcTpcHit*>::const_iterator iterHit = hits.begin();
                iterHit != hits.end();
                iterHit++)
              {
               StMcTpcHit*hit=*iterHit;
               if (!hit)
                    throw runtime_error("StiKalmanTrackFinder::loadMcHits() -E-
hit==0");
               unsigned int row = hit->padrow()-1;
               unsigned int sector = hit->sector()-1;
               unsigned int stiSector;
               if (sector<12)
                    stiSector = sector;
               else
                    stiSector = 11 - (sector-11)%12;
               StiDetector * detector = _detector->getDetector(row,stiSector);
               if (!detector)
                     throw runtime_error("StiKalmanTrackFinder::loadMcHits() -E-
Detector element not found");
               StiHit * stiHit = _hitFactory->getInstance();
               if(!stiHit)
                     throw runtime_error("StiKalmanTrackFinder::loadMcHits() -E-
stiHit==0");
               stiHit->reset();
               stiHit->setGlobal(detector,
                                 0,
                                 hit->position().x(),
                                 hit->position().y(),
                                 hit->position().z(),
                                 hit->dE());
               if (useMcAsRec)
```

```
                    _hitContainer->push_back( stiHit );
                }
            }
    //cout << "StiKalmanTrackFinder::loadMcHits() -I- Track done"<<endl;
    }
}
```

Figure 9-12 Examle Implementation of the loadMcHits(…) method.

9.2.4 Class StiXxxIsActiveFunctor

A StiXxxIsActiveFunctor is used to determine whether specific detector modules or subcomponents are to be considered active, i.e. bearing hit measurements. This is useful to handle detector components which are physically present during the experiment but that may have failed during an extended period of time. A StiXxxIsActiveFunctor is a class defined as a functor inheriting from the base class StiIsActiveFunctor. As such the derived class must implement the method "bool operator()(double y, double z)". This method should return true if the point (y,z) is within the active area of the detector and false otherwise. The functor is called by the "isActive(double y, double z)" method of the StiDetector class.

A class StiNeverActiveFunctor is available in the Sti package to set the state of detector components that are always passive. Example of such components include the beam pipe and the inner field cage.

The implementation of the StiXxxIsActiveFunctor depends on the specifics of the Xxx detector. Here we provide an example based on the TPC implementation. The header file is shown in Figure 9-14. The class features a constructor, a destructor, and the operator() method. The constructor implementation is shown in whereas the "bool operator()(double dYlocal, double dZlocal)" is presented in Figure 9-15. The class also defines three protected data members: s_pRdoMasks, m_bEastActive, and m_bWestActive. s_pRdoMasks is a static pointer to a StDetectorDbTpcRDOMasks structure which is pulled out of the TPC database. It is used to fetch RDO masks, and set the values of the boolean flags m_bEastActive and m_bWestActive which when true indicate the east and west sectors are active.

```
#include "Sti/StiIsActiveFunctor.h"
class StDetectorDbTpcRDOMasks;

class StiTpcIsActiveFunctor : public StiIsActiveFunctor{
  public:
    /// construct an IsActiveFunctor representing one TPC padrow
    /// spanning both TPC halves.  The sector should be [1-12],
    /// based on the half in the west TPC.  Padrow is in [1-45].
    StiTpcIsActiveFunctor(int iSector, int iPadrow);
    virtual ~StiTpcIsActiveFunctor();
    virtual bool operator()(double dYlocal, double dZlocal);

protected:
   /// returns the RDO board number [1-6] for the tpc padrow [1-45]
   inline static int rdoForPadrow(int iPadrow);

   /// pointer to instance of RDO mask
   static StDetectorDbTpcRDOMasks *s_pRdoMasks;

   /// is the east half of the padrow on?
   bool m_bEastActive;
   /// is the west half of the padrow on?
   bool m_bWestActive;
};

///Function returns the rdo board number for a given
///padrow index.
///Range of map used is 1-45.
int StiTpcIsActiveFunctor::rdoForPadrow(int iPadrow)
{
  int iRdo = 0;
  if (iPadrow>0&&iPadrow<=8)     iRdo = 1;
  else if (iPadrow>8&&iPadrow<=13)     iRdo = 2;
  else if (iPadrow>13&&iPadrow<=21)    iRdo = 3;
  else if (iPadrow>21&&iPadrow<=29)    iRdo = 4;
  else if (iPadrow>29&&iPadrow<=37)    iRdo = 5;
  else if (iPadrow>37&&iPadrow<=45)    iRdo = 6;
  return iRdo;
}
```

Figure 9-13 Example of a header file definition of the StiTpcIsActiveFunctor

The StiTpcIsActiveFunctor header file, shown in, also provides the implementation of the inlined static int rdoForPadrow(int iPadrow) which returns the readout board index (rdo) for any given pad row index.

The determination of the state of any given detector element is performed in the class constructor. The StiTpcIsActiveFunctor constructor code is shown in Figure 9-14 to illustrate the steps required to fetch the mask information from the star tpc database and set the m_bWestActive and m_bEastActive appropriately. There is one instance of the functor created for each sector and pad row combination. The constructor thus takes the sector and padrow to be set as argument. Note that the Sti index convention assumes indices to start at zero (0) whereas the STAR hardware indices start at 1. The sector and row indices are thus incremented by one internally to match the hardware definition (highlighted in yellow). One determines the index of the relevant readout board with the mapping function rdoForPadrow(int). One throws a runtime exception if the rdo_mask structure is not available. The flags m_bWestActive and m_bEastActive are set on the basis of the values returned by the "isOn" method of the rdo mask (high-lighted in red).

```
StiTpcIsActiveFunctor::StiTpcIsActiveFunctor(int iSector, int iPadrow)
{
  if(s_pRdoMasks==0)
    s_pRdoMasks = StDetectorDbTpcRDOMasks::instance();
  // STI to Hardware translation
  ++iSector;
  ++iPadrow;
  int iRdo = rdoForPadrow(iPadrow);
  if (!s_pRdoMasks)
    throw runtime_error("StiTpcIsActiveFunctor::StiTpcIsActiveFunctor(...) -
E- s_pRdoMasks==0");
  m_bWestActive = s_pRdoMasks->isOn(iSector, iRdo);
  m_bEastActive = s_pRdoMasks->isOn(24 - iSector%12, iRdo);
}
```

Figure 9-14 Example implementation of the StiXxxIsActiveFunctor constructor

Figure 9-15 presents an example of implementation of the bool StiTpcIsActiveFunctor:: operator()(double dYlocal, double dZlocal). Here one neglects the "y" information and use the position in Z (along the drift direction in the TPC) as well as the east and west flags to determine whether the given point is inside an active region of the detector.

```
bool StiTpcIsActiveFunctor::operator()(double dYlocal, double dZlocal)
{
  if (dZlocal<0.)
    return m_bWestActive && dZlocal>=-200.0;
  else
    return m_bEastActive && dZlocal<= 200.0;
}
```

Figure 9-15 Example implementation of the operator()(double dYlocal, double dZlocal) method for the TPC.

9.2.5 Class StiXxxHitErrorCalculator

The class StiXxxHitErrorCalculator is required to calculate estimates of hit errors based on the kinematic attributes of tracks. As an example, consider that the TPC hit errors are found to depend on the track crossing angle, dip angle, as well as the drift distance. Different detectors such as the TPC, the SVT, and the FTPC have different characteristics and resolutions, their hit errors must then be parametrized differently.

StiXxxHitErrorCalculator must be implemented as a derived class of the abstract class StiHitErrorCalculator. The StiHitErrorCalculator class definition is shown in Figure 9-16. The constructor and destructor of this class perform no operations. The class defines one pure virtual method called calculateError(). This method must be implemented by concrete class based on this abstract class to perform an actual calculation of hit erros for specific detectors.

```
class StiHitErrorCalculator
{
 public:
  StiHitErrorCalculator(){/*noop*/};
  virtual ~StiHitErrorCalculator(){/*noop*/};
  virtual void calculateError(StiKalmanTrackNode *)const = 0;
};
```

Figure 9-16 Definition of the StiHitErrorCalculator abstract class.

We illustrate the implementation of the StiHitErrorCalculator interface for the TPC. Figure 9-17shows the header file of the StiDefaultHitError calculator (It should really be called StiTpcHitErrorCalculator…). The constructor and destructor perform no operation. The method "set" must be used to initialize the coefficients "coeff" of the calculator. The actual error calculation is performed by the "calculateError" method on the basis of the parameters of the track node supplied as an argument.

```
class StiDefaultHitErrorCalculator: public StiHitErrorCalculator
{
 public:
   StiDefaultHitErrorCalculator();
   ~StiDefaultHitErrorCalculator();
   inline void calculateError(StiKalmanTrackNode *node) const;
   inline void set(double intrinsicZ, double driftZ,
                   double crossZ, double intrinsicX,
                   double driftX, double crossX);

 private:
   void  SetSource(int);
   int   Source;   //-1=error, 0=Default, 1=IOBroker, 2=User Defined
   double coeff[6];  //0:intrinsicZ  1: driftZ   2: crossZ
                 //3:intrinsicX  4: driftX   5: crossX
};
```

Figure 9-17 Definition of the StiDefaultHitErrorCalculator concrete class.

The implementation of the calculateError method is supplied in Figure 9-18 to illustrate how one calculates the TPC hit errors on the basis of the position along the drift direction and the crossing and dip angles of the given track. Note that the calculateError method has no return value and sets the values of the "eyy" and "ezz" data members of the Kalman track node (highlighted in yellow in Figure 9-18). These two variables correspond to the square of the error along the pad and drift directions respectively. They are used in the determination of the track chi-square as well for the determination of the hit search cone size.

The parameterization used in this method is that of Blum & Rolandi (see Mike Lisa's web site at http://www.star.bnl.gov/~lisa/HitErrors/).

$\sigma = \{ \sigma^2_{int} + + \sigma^2_{diff} * d_{drift}/\cos^2(\alpha \text{ or or}\lambda) + \sigma^2_{cros} * \tan^2(\alpha \text{ or or}\lambda)\}^{1/2}$

for both the errors along the pad direction (ecross in the code) and the drift direction (edip in the code).

Note that scaling parameters are used to enable fine tuning and debugging from a graphical user interface (as highlighted in blue).

```
void   StiDefaultHitErrorCalculator::calculateError(StiKalmanTrackNode   *
node) const
{
  double dz = (fabs(node->getZ())-200.)/100.;
  double cosCA = node->_cosCA;
  double sinCA = node->_sinCA;
  if (cosCA==0.)     cosCA=1.e-10;
  double tanCA = sinCA/cosCA;
  double                         ecross=coeff[0]+coeff[1]*dz/(cosCA*cosCA)
+coeff[2]*tanCA*tanCA;
  double tanDip=node->getTanL();
  double cosDipInv2=1+tanDip*tanDip;
  double edip=coeff[3]+coeff[4]*dz*cosDipInv2+coeff[5]*tanDip*tanDip;
  if (ecross>50) ecross = 50.;
  if (edip>50) edip = 50.;
  double scaling;
  if (node->_x>120)
    scaling = StiKalmanTrackNode::pars->getOuterScaling();
  else
    scaling = StiKalmanTrackNode::pars->getInnerScaling();
  node->eyy = ecross*scaling*scaling; // in cm^2
  node->ezz = edip*scaling*scaling;
}
```

Figure 9-18 Implementation of the calculateError() method for the TPC.

9.3 Modifications to StiMaker/macros/Run.C

The ITTF tracker can be launched with a simple macro Run.C provided in the package StiMaker. Detailed documentation of this macro is provided in XXXX. Here, we list the changes to be done to the macro in order to include a new detector group.

The macro carries two arguments for each detector group used in the tracker. They are respectively called "useXXX" and "activeXXX". Both are Boolean parameters. The "useXXX" specifies whether the given detector group XXX shall be used and instantiated. The "activeXxx" specifies whether the detector group Xxx is considered active (useXXX==true) or passive (useXXX=false) while running the tracker. Arguments for a new detector group should be added sequentially after the existing detector groups.

Modifications to StiMaker/StiMakerParameters

The StiMakerParameters class carries all settable attributes used by the StiMaker. These include, in particular, "useXXX" and "activeXxx" Boolean flags for each detector group to be used in the tracker. New entries shall be added sequentially in the class header file.

Modifications to StiMaker/StiMaker

The decision to use and instantiate a specific detector group is taken at run time within the StiMaker on the basis of the “useXXX” attributes carried by StiMakerParameters and initialized with the Run.C macro. The “useXXX” Boolean variable determines whether the Xxx detector group shall be instantiated and added to list of groups used by the tracker. This is accomplished by first instantiating the group class (e.g. StiXxxDetectorGroup) and adding the instance to the list of such instances held by the tracker master toolkit.

Figure 9-19 presents an example of the code needed to perform such an operation.

```
Int_t StiMaker::InitDetectors()
{
 StiDetectorGroup<StEvent,StMcEvent> * group;
 cout<<”StiMaker::InitDetectors() -I- Adding detector group:Star”<<endl;
 _toolkit->add(new StiStarDetectorGroup());
 if (_pars->useTpc)
   {
    cout<<”StiMaker::InitDetectors() -I- Adding detector group:TPC”<<endl;
    _toolkit->add(group = new StiTpcDetectorGroup(_pars->activeTpc));
    group->setGroupId(kTpcId);
   }
 if (_pars->useSvt)
   {
    cout<<”StiMaker::Init() -I- Adding detector group:SVT”<<endl;
    _toolkit->add(group = new StiSvtDetectorGroup(_pars->activeSvt));
    group->setGroupId(kSvtId);
   }
 if (_pars->useFtpc)
   {
    cout<<”StiMaker::Init() -I- Adding detector group:FTPC”<<endl;
    _toolkit->add(group = new StiFtpcDetectorGroup(_pars->activeFtpc));
    group->setGroupId(kFtpcWestId);
   }
 if (_pars->useEmc)
   {
    cout<<”StiMaker::Init() -I- Adding detector group:BEMC”<<endl;
    _toolkit->add(group = new StiEmcDetectorGroup(_pars->activeEmc));
    group->setGroupId(kBarrelEmcTowerId);
   }
 return kStOk;
}
```

Figure 9-19 Method InitDetector() of the StiMaker class to be modified for the addition or removal of detector groups.

In this example, one instantiates and sequentially adds the STAR basic components (e.g. beam pipe), the TPC, SVT, FTPC and EMC, provided the corresponding "_pars" flag are set to true.

Note that after the group is instantiated, one must sets its identifier to an agreed upon integer value. This value can then be used within the tracker to fetch or identify detector elements belonging to this group.

Note additionally that the instance of the group internally takes care of instantiating all ancillary objects and classes required by this specific group (See documentation in 9.2.1).

10 Track Finder and Fitter

10.1 Introduction

The ITTF tracker consists of three essential components: a track seed finder, a track finder (or follower), and a track fitter. Given there are in principle many ways in which one can implement these three components, we defined three abstract interfaces to define their attributes, behavior, and formalized their definition. The abstract interfaces are presented in Section 10.2. The seed finder implementation is discussed first in Section 10.3. The track finder and track fitter implementations are presented in Sections 10.4 and 0 respectively.

10.2 Abstract Class Definitions

The notions of track seed finder, track finder (or follower), and track fitter are formalized with the definition of abstract interface (pure virtual classes) presented in the next three subsections.

10.2.1 StiTrackSeedFinder Class

A track seed finder is formally defined with the abstract class StiTrackSeedFinder presented in Figure 10-1. The definition is rather simple and straightforward. Almost all methods are pure virtual and should therefore be implemented in a derived class. They define the expected behavior of a track seed finder.

The role of the seed finder is to find track seeds on the basis of unused hits present in the input StiHitContainer store. A seed finder shall be recursively callable from a track finder to initiate tracks, i.e. to find track stubs that can be used to initiate formal track searches. Many types and instances of seed finders might be used concurrently in the tracker, It is thus convenient to have this class nameable and effectively inherit from the Named class.
It is also convenient to have seed finder parameters encapsulated as editable parameters (using the EditableParameters class). The abstract seed finder definition currently in use defines the seed finder to inherit from the EditableParameters. We foresee a modification of the class whereby the "has parameters" pattern shall be implemented by composition rather than inheritance.
A seed finder requires access to a hit container, a track factory, and a detector container. These items shall be set at construction of a seed finder object or through the use of modifier methods *setFactory(...), setHitContainer(...),* and *setDetectorContainer().*

This class is abstract given many (most) of its methods are pure virtual. These methods shall be implemented in a concrete class deriving this class.
The *hasMore()* method shall be implemented to indicate whether the seed finder currently has more track seeds available. Note that this should not be a contractual promise. The expected behavior is that the method shall return false if all possibilities of forming have been exhausted. It shall return true if seed might still be generated on the basis of the hits left in the hit container.
The *next()* method shall be implemented to find the next possible track seed and return a pointer to it to the caller. Note that it shall be possible for this method to return a null pointer if the search for a seed failed. It shall also be acceptable for the *hasMore()* and *next()* methods to sequentially return true and a null pointer given one does not wish to require the hasMore() method to effect a search but rather to indicate the possible outcome of a search (with no false negatives, but possibly false positives).
The *reset()* method shall be implemented to perform an internal reset of the seed finder.

```
class StiTrackSeedFinder : public EditableParameters
{
public:
  StiTrackSeedFinder(const string& name,
                     Factory<StiKalmanTrack> * trackFactory,
                     StiHitContainer         * hitContainer,
                     StiDetectorContainer    * detectorContainer);
  virtual ~StiTrackSeedFinder();
  virtual bool hasMore() = 0;
  virtual StiKalmanTrack* next() = 0;
  virtual void reset() =0;
  virtual EditableParameters * getParameters();
  void setFactory(Factory<StiKalmanTrack>* val);
  void setHitContainer(StiHitContainer*);
  StiHitContainer* getHitContainer();

 protected:
  Factory<StiKalmanTrack>* _trackFactory;
  StiHitContainer* _hitContainer;
  StiDetectorContainer* _detectorContainer;
  Messenger &  _messenger;

private:
  StiTrackSeedFinder(); //Not implemented
};
```

Figure 10-1 Definition of the StiTrackSeedFinder Class.

10.2.2 StiTrackFinder Class

A track finder is formally defined with the abstract class StiTrackFinder presented in Figure 10-2. The definition is rather simple and straightforward. All methods are pure virtual and should therefore be implemented in a derived class. They define the expected behavior of a tracker.
Method *initialize()* shall be implemented to initialize all components of the tracker.
Method *reset()* shall be implemented to reset and clear the output of the tracker so one can process track searches and fitting on the current event once again without reloading it.

Method *clear()* shall be implemented to reset and clear the input and output stores used by the tracker so a new event can be loaded and reconstructed.
Method *findTracks()* shall be implemented to initiate a search of all tracks that may be reconstructed in the current event. It may be called from a maker directing the track reconstruction or from am event display.
Method *findTrack(StiTrack* track, int direction)* shall be implemented to initiate the extension (search) on the given track seed. The given direction may be set to be inside-out or inside-in. It shall be iteratively called by the *findTracks()* method implementation.
Method *findNextTrack()* shall be implemented to initiate the search and extension of a new track. It is essentially provided to permit track-by-track reconstruction and visualization from an event display.
Method *extendTracksToVertex(StiHit* vertex)* shall be implemented to iterate on all found tracks, and attempt their extension to the given vertex position.
Method *fitTracks()* shall be implemented to enable the tracker to invoke a track fitter to fit all currently found tracks.
Method *fitNextTrack()* shall be implemented to enable the tracker to fit or refit the next available track stored in the track container.
The accessor method *getTrackFilter()* shall be implemented to return a pointer to the track filter currently in use by the tracker.
The accessor method *getVertexFinder()* and modifier method *setVertexFinder (StiVertexFinder *)* shall be implemented to get and set the vertex finder used by the tracker.
The accessor method *getParameters*() shall be implemented to return an instance of the EditableParameters encapsulating all editable and settable parameters of the tracker.

```
class StiTrackFinder
{
public:
  /// Initialize the finder
  virtual void initialize()=0;
  /// Find all tracks of the current event
  virtual void findTracks()=0;
  /// Find/extend the given track, in the given direction
  virtual bool find(StiTrack *track, int direction) = 0;
  /// Find the next track
  virtual void findNextTrack()=0;
  /// Fit all tracks currently loaded
  virtual void fitTracks()=0;
  /// Fit the next track available
  virtual void fitNextTrack()=0;
  /// Extent all tracks to the given vertex
  virtual void extendTracksToVertex(StiHit* vertex)=0;
  /// Reset the tracker
  virtual void reset()=0;
  /// Clear all stores of the tracker
  virtual void clear()=0;
  /// Get the track filter currently used by the tracker
  virtual Filter<StiTrack> * getTrackFilter() const = 0;
  /// Get the vertex finder used by this track finder
  virtual StiVertexFinder * getVertexFinder()=0;
  /// Set the vertex finder used by this tracker
  virtual void setVertexFinder(StiVertexFinder *)=0;
  /// Get the track parameters
  virtual EditableParameters * getParameters()=0;
};
```

Figure 10-2 Definition of the StiTrackFinder Class.

10.2.3 StiTrackFitter Class

A track fitter is formally defined with the abstract class StiTrackFitter presented in Figure 10-2. The definition is extremely simple: it features a single pure virtual method named *fit*.
The *fit(StiTrack * track, int direction)* method shall be implemented in a derived class to perform a fit or refit (as appropriate) of the given track in the given direction.

```
class StiTrackFitter
{
public:
    StiTrackFitter();
    virtual ~StiTrackFitter();
    virtual void fit(StiTrack * track, int direction)=0;
};
```

Figure 10-3 Definition of the StiTrackFitter Class.

10.3 Track Seed Finder

Aware that many track seed finding scenarios exist, we have implemented the track seed finder to derive from the abstract base class presented in Section 10.2.1. In fact, we actually created two seed finders. The first is an actual finder operating on the basis of a local combinatorial follow your nose type algorithm. The second finder is a composite finder or broker capable of storing pointers to multiple finders, and calling them in sequence to find and deliver track seeds to the tracker. One can thus use seed finders with different capabilities or operated in different parts of the detector e.g. central TPC vs forward TPCs.
The composite track seed finder StiCompositeTrackSeedFinder is discussed first in Section 10.3.1. The local seed finder currently in use in the tracker is presented next in Section 10.3.2.

10.3.1 StiCompositeTrackFinder Class

The StiCompositeTrackFinder class is defined with public inheritance on the StiTrackSeedFinedr abstract class and the STL vector class with template argument StiTrackSeedFinder. It is a seed finder broker: It behaves as a finder but uses arbitrarily many elementary registered finders to actually generate track seeds.
The methods initialize(), hasMore(), next(), reset() are implemented to fulfill their abstract base class definition. In all four cases, an internal loop is used to initialize, poll, reset, or fetch a seed from the elementary seed finders registered with the composite finder. Elementary seed finders can be registered with the composite finder by calling the push_back() method of the vector<StiTrackSeedFinder*> base class.

```
class StiCompositeSeedFinder : public StiTrackSeedFinder, public
vector<StiTrackSeedFinder*>
{
public:
  StiCompositeSeedFinder(const string&            name,
                 Factory<StiKalmanTrack>* trackFactory,
                 StiHitContainer*         hitContainer,
                 StiDetectorContainer*    detectorContainer);
  virtual ~StiCompositeSeedFinder();
  void    initialize();
  virtual bool hasMore();
  virtual StiKalmanTrack* next();
  virtual void reset();
 protected:
  vector<StiTrackSeedFinder*>::iterator _currentTrackSeedFinder;
 private:
  //Not implemented
  StiCompositeSeedFinder();
};
```

Figure 10-4 Definition of the StiCompositeTrackFinder Class.

The implementation of the next() method of the StiCompositeTrackFinder class is shown in Figure 10-5 for illustrative purposes. An iterator named *_currentTrackSeedFinder* is used to keep track of the current or active seed finder. The iterator is dereferenced to obtain the current finder. The next() method of that finder is then called, and its return value shall be return by this method. However, before returning, one checks if the current finder has more seeds. If not, the iterator is incremented to use the next finder available during subsequent calls to this method.

```
StiKalmanTrack* StiCompositeSeedFinder::next()
{
  StiKalmanTrack* track=0;
  track = (*_currentTrackSeedFinder)->next();
  //Check to see if we ran out
  if ( (*_currentTrackSeedFinder)->hasMore()==false )
      ++_currentTrackSeedFinder;
  return track;
}
```

Figure 10-5 Implementation of the next() method of the StiCompositeTrackFinder class

10.3.2 StiLocalTrackFinder Class

```
class StiLocalTrackSeedFinder : public StiTrackSeedFinder
{
public:
  StiLocalTrackSeedFinder(const string& name,
                    Factory<StiKalmanTrack> * trackFactory,
                    StiHitContainer         * hitContainer,
                    StiDetectorContainer    * detectorContainer);
  virtual ~StiLocalTrackSeedFinder();
  virtual bool hasMore();
  virtual StiKalmanTrack* next();
  virtual void reset();
  virtual void initialize();
  virtual void addLayer(StiDetector*);
  virtual void print() const;

protected:
  StiSortedHitIterator begin();
  StiSortedHitIterator end();
  ///Extend hit looking for closest neighbor in z
  bool extendHit(StiHit * hit);
  ///Extrapolate to next layer using straight line, add hit closest in z
  bool extrapolate();
  StiKalmanTrack* initializeTrack(StiKalmanTrack*);
  void calculate(StiKalmanTrack*);
  void calculateWithOrigin(StiKalmanTrack*);
  //Perform helix fit, Perform helix calculation (doesn’t assume any
vertex)
  bool fit(StiKalmanTrack*);
  typedef vector<StiHit*> HitVec;
  typedef vector<StiDetector*> DetVec;
 StiSortedHitIterator _hitIter;
  virtual StiKalmanTrack* makeTrack(StiHit*);
  DetVec mDetVec;

  //define search window in the next layer when connecting two points
  double mDeltaY;
  double mDeltaZ;
  //define the number of points to connect
  int mSeedLength;
  //define search window in the next layer when extending a coonection of
points
  double mExtrapDeltaY;
  double mExtrapDeltaZ;
  //Count how many hits we’ve skipped in extrapolation
  int mSkipped;
  //Define the max number we can skip
  int mMaxSkipped;
  //define the Min/Max number of points to extrapolate
  int mExtrapMinLength;
  int mExtrapMaxLength;
  //Use the origin to calculate helix?
  bool mUseOrigin;
  vector<StiHit*> mSeedHitVec;
  bool mDoHelixFit; //true-> fit, false-> calculate
  StiHelixCalculator mHelixCalculator;
  StiHelixFitter mHelixFitter;

 private:
  //The following are not implemented, as they are non-trivial
  //and the default compiler generated versions will be wrong.
  StiLocalTrackSeedFinder();
  StiLocalTrackSeedFinder(const StiLocalTrackSeedFinder&);
  StiLocalTrackSeedFinder operator=(const StiLocalTrackSeedFinder&);
};
```

Figure 10-6 Definition of the StiLocalTrackFinder Class.

10.4 Track Finder

The ITTF tracker finder is implemented by the StiKalmanTrackFinder class.
The class definition is shown in Figure 10-7. It is a concrete class which inherits from the abstract interface StiTrackFinder. As such, it implements all pure virtual methods of that class with their intended interpretation and behavior. This is a big and somewhat busy class. And it is the main class of the tracker, a detail discussion of its members and methods is in order…

```
class StiKalmanTrackFinder : public StiTrackFinder
{
public:
  StiKalmanTrackFinder(StiToolkit*toolkit);
  virtual ~StiKalmanTrackFinder();
  /// Initialize the finder
  virtual void initialize();
  /// Find all tracks of the currently loaded event
  virtual void findTracks();
  /// Find/extend the given track, in the given direction
  virtual bool find(StiTrack *track, int direction);
  /// Find the next track
  virtual void findNextTrack();
  /// Find the next track segment
  virtual void findNextTrackSegment();
  /// Fit all tracks crruently loaded
  virtual void fitTracks();
  /// Fit the next track available
  virtual void fitNextTrack();
  /// Extent all tracks to the given vertex
  virtual void extendTracksToVertex(StiHit* vertex);
  /// Reset the tracker
  virtual void reset();
  //virtual void update();
  /// Clear the tracker
  virtual void clear();
  /// Get the number of number of track seed used by the seed finder
  virtual int getTrackSeedFoundCount() const;
  /// Get the number of track found
  virtual int getTrackFoundCount() const;
  /// Get the number of track found that satisfy the given filter
  virtual int getTrackFoundCount(Filter<StiTrack> * filter) const;
  /// Get the track filter currently used by the tracker
  virtual Filter<StiTrack> * getTrackFilter() const;

  /// Get the vertex finder used by this track finder
  virtual StiVertexFinder * getVertexFinder();
  /// Set the vertex finder used by this tracker
  virtual void setVertexFinder(StiVertexFinder *);

  /// Set Tracking Mode used for Interactive Tracking
  void setTrackingMode(StiFindStep m);
  /// Get Tracking Mode used for Interactive Tracking
  StiFindStep getTrackingMode() const;

  void setParameters(StiKalmanTrackFinderParameters *par);
  virtual EditableParameters * getParameters();

  void doInitLayer(int trackingDirection);
  void doNextDetector();
  void doFinishLayer();
  void doFinishTrackSearch();
  void doNextTrackStep();
protected:
   StiToolkit                * _toolkit;
   Filter<StiTrack>           * _trackFilter;
   StiTrackSeedFinder         * _trackSeedFinder;
   Factory<StiKalmanTrackNode> * _trackNodeFactory;
   Factory<StiKalmanTrack>     * _trackFactory;
   Factory<StiMcTrack>        * _mcTrackFactory;
   Factory<StiHit>            * _hitFactory;
   StiDetectorContainer       * _detectorContainer;
   StiHitLoader<StEvent,StMcEvent,StiDetectorBuilder> * _hitLoader;
   StiHitContainer            * _hitContainer;
   StiTrackContainer          * _trackContainer;
   StiTrackContainer          * _mcTrackContainer;
   StiVertexFinder            * _vertexFinder;
   StiStEventFiller           * _eventFiller;
   StEvent                   * _event;
   StMcEvent                  * _mcEvent;
   StiKalmanTrackFinderParameters * _pars;
   Messenger                  & _messenger;
// private data members are omitted for clarity…
};
```

Figure 10-7 Definition of the StiKalmanTrackFinder Class.

10.4.1.1 PROTECTED DATA MEMBERS

The class requires and relies on a host of external entities to perfrom its task. Protected data members are thus used to store pointers to those various entities.
Access to the StiToolkit is need at initialization time to obtain all other components required by the tracker. For efficiency reasons, one thus keep a pointer to it *(_toolkit*)as a protected data member.

The tracker uses track nodes and reconstruct tracks. It thus requires to frequently gain access to instances of StiKalmanTrackNode and StiKalmanTrack. Pointers to the node and track factories are thus also kept as protected data menber of the class.
For the same efficiency reasons, one stores pointers to the hit container (_hitContainer) used as hit input, the track container (_trackContainer) used to store produced tracks, the seed finder (_trackSeedFinder) to initiate tracks, etc. _trackFilter points to a filter used to judge whether reconstructed tracks pass a minimal quality cut, should be stored in the track container, and their hits declared as used. _detectorContainer points to the container holding the geometry of the detector. This detector container is used in the propagation of the tracks to fetch detector objects and account for their physical properties for the purpose of calculating MCS and energy loss effects.
_vertexFinder is a pointer to the vertex finder to be used to find the proimary vertex once global tracks have been reconstructed.
_eventFiller is a pointer to the StiStEventFiller object used to convert StiTracks data into the StEvent STAR persistent data model.
_pars is a pointer to the StiKalmanTrackFinderParameters object used to hold the parameters of the tracker.

10.4.1.2 PUBLIC METHODS

Methods of the base class StiTrackFinder are implemented in the StiKalmanTrackFinder to provide their intended functionality. We focus the presentation on the *findTracks()* and *findTrack(StiKalmanTrack*)* methods.

The core of the *findTracks()* method is displayed in Figure 10-8. A while loop predicated on the output of the track seed finder hasMore method is used to find all possible tracks. A try/catch block is used to trap run time errors that may occur in the propagation of a track. One first extract obtain a track seed for the track seed finder by calling its next function. If a null pointer is returned, it means the finder has exhausted all seed possibilities, one thus exits the loop with the break statement. If a valid pointer is returned, one proceeds to tell the track to find itself by calling its find method. Once the find method returns, one proceeds to test the found track to verify whether it meets the minimum quality requirements to be stored and its hits declared as used. The track is stored by a call to the push_back method of the track container _trackContainer. The hits associated with the tracks are declared as used with a call to the reserveHits method of the track.

```
while (_trackSeedFinder->hasMore())
        {
         try
           {
            // obtain track seed from seed finder
            track = _trackSeedFinder->next();
            // a null pointer there are no more seeds left
            if (!track) break;
            // tell the track to find itself
            track->find();
            // if the filter is defined and the track is acceptable, store it and
        // reserve the hits.
            if (!_trackFilter || _trackFilter->filter(track))
                 {
                  _trackContainer->push_back(track);
                  track->reserveHits();
                 }
           }
         catch (runtime_error & rte)
           {
            cout<< "StiKalmanTrackFinder::findTracks() - Run Time Error :"
              << rte.what() << endl;
           }
        }
```

Figure 10-8 Core of the findTracks method of the StiKalmanTrackFinder class.

The implementation of the find method of the StiKalmanTrack is shown in abreviated form in Figure 10-9 and described next. Try/catch block and error message streams have been removed for clarity. One uses Boolean flags *trackExtended* and *trackExtendedOut* to keep track of whether the track is successfully extended inward and outward respectively. The track flag is initially set to zero to signify that the track reconstruction is in progress. It is eventually set to one to identify the track as succesfully reconstructed. One first invokes the find method of the trackFinder with this track and the kOutsideIn direction indicator. A successful extension of the track returns a true value. One then proceeds to fit in the track in the reverse direction. This allows to properly propagate the Kalman state and error information to the outer most node of the track. Note that the first 3 to 6 nodes of the track are initialize with a rough guess of the momentum and errors. Their state must then be properly updated to account for the information gained by tracking and propagating the track inward. This is done a by a call to the fit(kInsideOut) method. If both the inward find and outward fit succeed, the track is declared extended.
The next step is to decide whether one should attempt to "find" an extension of the track outward, starting from the current outward most node on the track. The decision is positive if the outward most node of the track is a radial position less than 190 cm. If the decision is positive, one first swap or reverse the track, and then call the trackFinder find method with the kInsideOut direction to extend the track outward. If the extension is successful (i.e.returns true), one must

refit the track inward to account for the hits added on the track. The track is finally swap back in its original state. The method then returns a true value if either of the inward or outward track extension were succesful.

```
bool StiKalmanTrack::find(int direction)
{
  bool trackExtended=false;
  bool trackExtendedOut=false;
  setFlag(0);
  // invoke tracker to find or extend this track
  if (trackFinder->find(this,kOutsideIn))
          {
            fit(kInsideOut); trackExtended = true;
          }
  // decide if an outward pass is needed.
  const StiKalmanTrackNode * outerMostNode = getOuterMostHitNode();
  if (outerMostNode->getX()<190. )
    {
      swap();
      trackExtendedOut= trackFinder->find(this,kInsideOut);
      if (trackExtendedOut) fit(kOutsideIn);
      swap();
    }
  setFlag(1);
  return trackExtended||trackExtendedOut;
}
```

Figure 10-9 Core of the find method of the StiKalmanTrack class.

10.4.2 Track Fitter

The ITTF tracker uses a track (re) fitter based on the Kalman filter methods of the StiKalmanTrackNode class. The fitter is implemented by the StiKalmanTrackFitter concrete class shown in Figure 10-10. StiKalmanTrackFitter inherits from the abstract interface StiTrackFitter. It implements the fit method to perform of Kalman tracks.

```
class StiKalmanTrackFitter : public StiTrackFitter
{
public:
   StiKalmanTrackFitter();
   virtual ~StiKalmanTrackFitter();
   virtual void fit(StiTrack * track, int direction);
};
```

Figure 10-10 Definition of the StiKalmanTrackFitter Class.

The core of the fitting procedure is illustrated in Figure 10-11. One uses a Kalman Track Node iterator (class StiKTNBidirectionalIterator) called source to iterate through all the nodes of the track to be (re)fitted. The same process is repeated iteratively. Given a node, one gets its child,

i.e. the next node on the track. One extracts the detector and hit associated with that node. If a detector is associated with the node, it is a normal hit, and one propagates the track from the current node (source) to the detector of the child node (target) using the propagate(StiKalmanTrackNode*, StiDetector*) method of the StiKalmanTrackNode class. If the target node has no associated detector, then it is a vertex, one then propagate the current track to the vertex position using the propagate(StiKalmanTrackNode*, StiHit*) method of the StiKalmanTrackNode class instead. Subsequently, if the targetNode holds a hit (be it a normal hit or a vertex), one uses this measured hit to update the Kalman state vector of the track by calling in sequence the evaluateChi2() and updateNode() methods of the StiKalmanTrackNode class on the target node. One finally increment the source iterator to repeat this process untill all nodes of the track have been traversed.
Note that the code fragment shown in Figure 10-11 illustrates the fitting procedure used when the fit is performed in the same direction as the tracking was performed. It is however sometimes necessary to do a fit in the reverse direction. The code used in that case is quite similar to the one shown here. The difference lies in the fact that one uses the parent of a node rather its child.

```
StiKTNBidirectionalIterator first;
StiKTNBidirectionalIterator last;
StiKTNBidirectionalIterator source;
double chi2;
first = track->begin();
last  = track->end();
for (source=first;source!=last;)          {
            targetNode= static_cast<StiKalmanTrackNode*>((*source).getFirstChild());
            targetDet = targetNode->getDetector();
            targetHit = targetNode->getHit();
            // evolve state from that of source using dets source to target
            if (targetDet) // hit case
              status = targetNode->propagate(&(*source),targetDet);
            else // vertex case
              status = targetNode->propagate(&(*source),targetHit);
            // if targetNode has hit, get chi2 and update track parameters accordingly
            if (targetHit && status==0)     {
               chi2 = targetNode->evaluateChi2(targetHit);
               targetNode->setChi2(chi2);
               targetNode->updateNode();
              }
       source++;
          }
   }
```

Figure 10-11 Core of the fit method: Iteration on track nodes.

11 Sti Toolkit

The STI toolkit provides a centralized mechanism to build and retrieve all components required by the tracker.

We have defined an abstract toolkit interface StiToolkit, which defines the (intended) behavior of the toolkit. The StiToolkit class definition is shown in Figure 11-1.

The actual implementation of the toolkit, called StiDefaultToolkit, is part of the StiMaker package. We refer the reader to the online documentation for details on that class.

StiToolkit is a singleton class; only one instance of this class should and can be instantiated. Its constructor is thus private. One can gain access to the toolkit by invoking the static method instance().

The class defines a host of accessor and modifier methods to get and set the multiple components required by the tracker. The interpretation of these method is for the most part self explanatory but comments are nonetheless inserted in yellow in Figure 11-1.

```
class StiToolkit  {
public:
/// Factory Accessors
/// Get Hit Factory
  virtual Factory<StiHit> * getHitFactory()=0;
///  Get Kalman Track (reconstructed) Factory
  virtual Factory<StiKalmanTrack> * getTrackFactory()=0;
/// Get Monte Carlo Track Factory
  virtual Factory<StiMcTrack> * getMcTrackFactory()=0;
/// Get Track Node Factory
  virtual Factory<StiKalmanTrackNode> * getTrackNodeFactory()=0;
/// Get Detector Factory
  virtual Factory<StiDetector>  * getDetectorFactory()=0;
/// Get Composite Tree Node Factory
  virtual Factory<StiCompositeTreeNode<StiDetector> >  *
              getDetectorNodeFactory()=0;
  //Container Accessors.
  /// Get Master Detector Builder
  virtual StiMasterDetectorBuilder * getDetectorBuilder()=0;
  /// Get Detector Container
  virtual StiDetectorContainer  * getDetectorContainer()=0;
  /// Get Detector Groups
  virtual StiDetectorGroups<StEvent,StMcEvent> * getDetectorGroups()=0;
```

```
/// Get Reconstructed Hit Container
virtual StiHitContainer      * getHitContainer()=0;
/// Get Monte Carlo Hit Container.
virtual StiHitContainer      * getMcHitContainer()=0;
/// Get Reconstructed Track Container
virtual StiTrackContainer    * getTrackContainer()=0;
/// Get Monte Carlo Track Container
virtual StiTrackContainer    * getMcTrackContainer()=0;
/// Service and convenience class objects.
virtual StiDetectorFinder    * getDetectorFinder()=0;
virtual StiTrackSeedFinder   * getTrackSeedFinder()=0;
virtual StiTrackFinder       * getTrackFinder()=0;
virtual StiTrackFitter       * getTrackFitter()=0;
virtual StiTrackMerger       * getTrackMerger()=0;
virtual StiVertexFinder      * getVertexFinder()=0;
virtual StAssociationMaker   * getAssociationMaker()=0;
virtual StiMaker             * getStiMaker()=0;
virtual StiResidualCalculator * getResidualCalculator()=0;
virtual StiHitLoader<StEvent,StMcEvent,StiDetectorBuilder> * getHitLoader()=0;
virtual void setAssociationMaker(StAssociationMaker * a)=0;
virtual void setStiMaker(StiMaker* a)=0;
virtual void add(StiDetectorGroup<StEvent,StMcEvent>* detectorGroup)=0;
virtual void setGuiEnabled(bool )=0;
virtual bool isGuiEnabled() const=0;
virtual void setMcEnabled(bool)=0;
virtual bool isMcEnabled() const=0;
virtual void setEvaluatorEnabled(bool)=0;
virtual bool isEvaluatorEnabled() const=0;

virtual EditableFilter<StiHit>   * getLoaderHitFilter()=0;
virtual EditableFilter<StiTrack> * getLoaderTrackFilter()=0;
virtual EditableFilter<StiTrack> * getFinderTrackFilter()=0;
```

```
  virtual void setLoaderHitFilter(EditableFilter<StiHit>   *)=0;
  virtual void setLoaderTrackFilter(EditableFilter<StiTrack> *)=0;
  virtual void setFinderTrackFilter(EditableFilter<StiTrack> *)=0;

  static void setToolkit(StiToolkit*toolkit);
  static StiToolkit *instance();
  static void kill();

 protected:
  static StiToolkit * _instance;
};
```

Figure 11-1 Definition of the StiToolkit abstract interface.

12 Appendix 1: Kalman Filter Theory

The Kalman filter is a recursive technique that enables a computationally efficient solution to the least square fit problem. Applied to track finding, it implies an estimate of the track parameters is performed each time a new hit is added to the track. This presents the advantage of a much simpler matrix inversion (2 x 2 versus 2N x 2N). Additionally, one can use the estimated track parameters after the inclusion of a given point to predict the position of the next point.
A basic Kalman filter deals with linear system of equations. We use the notation of Fruhwirth, and use the following definitions:

$\mathbf{x_k}$	the filtered state vector at point k
$\mathbf{x_{k+1}}^{\mathbf{k}}$	the extrapolated state vector from point k to point k+1
$\mathbf{C_{k+1}}^{\mathbf{k}}$	the covariance matrix of $\mathbf{x_{k+1}}^{\mathbf{k}} - \mathbf{x_{k+1}}^{\mathbf{t}}$, where $\mathbf{x_{k+1}}^{\mathbf{t}}$ is the true value of the state vector at point k+1
$\mathbf{C_k}$	filtered covariance matrix at point k
$\mathbf{F_k}$	propagator of the state vector from point k to point k+1
$\mathbf{w_k}$	process noise (random disturbance) to the state vector at point k
$\mathbf{Q_k}$	covariance matrix of $\mathbf{w_k}$
$\mathbf{m_k}$	measurement vector at point k
$\mathbf{e_k}$	measurement noise at point k
$\mathbf{V_k}$	covariance matrix of $\mathbf{e_k}$

The two basic equations are:

$$\mathbf{x_k} = \mathbf{F_{k-1}}\, \mathbf{x_{k-1}} + \mathbf{w_{k-1}}$$
$$\mathbf{m_k} = \mathbf{H_k}\, \mathbf{x_k} + \mathbf{e_k},$$

where $\mathbf{x_0}$, is an initial estimate of the state vector, $\mathbf{F_k}$ is the state propagator matrix at the k^th point, $\mathbf{Q_k}$ the process noise covariance matrix evaluated at the k^th point, $\mathbf{V_k}$, the measurement noise covariance matrix, and $\mathbf{H_k}$ a matrix which converts the state vector at point k into a measurement vector, m_k.
The initial state covariance matrix $\mathbf{C_0}$ can be set to the identity matrix multiplied by a large-scale factor. The smaller this is, the more weight is put on the initial state vector, so in general we would like to make $\mathbf{C_0}$ as large as possible. However, because of round-off errors, one needs to restrict this matrix to a reasonable value, which depends on the particular fit under consideration.

The Kalman filter operation involves the following operation at each step of the track reconstruction:

$$C_k^{k-1} = F_{k-1}\, C_{k-1}\, F_{k-1}^{T} + Q_{k-1}$$
$$R_k^{k-1} = V_k + H_k\, C_k^{k-1}\, H_k^{T}$$
$$K_k = C_k^{k-1}\, H_k^{T}\, (\, R_k^{k-1}\,)^{-1}$$
$$r_k^{k-1} = m_k - H_k\, F_{k-1}\, x_{k-1}$$
$$x_k = F_{k-1}\, x_{k-1} + K_k\, r_k^{k-1}$$
$$C_k = (1 - K_k H_k)\, C_k^{k-1}$$
$$(\square^2)^k_{k-1} = (r_k^{k-1})^T\, (R_k^{k-1})^{-1}\, r_k^{k-1}$$

At the end of the iterative process, the final state vector $\mathbf{x_k}$ represents the fit values using all data points. To obtain the fit values at any point k, the user can then fit backwards, starting with the last point. The fit value at some point k is then the average between the state vectors $\mathbf{x_k}$ between the two fits. The goodness of the fit can be evaluated based on the chi-square χ^2 as evaluated in step 7 above.

13 Appendix 2: Multiple Scattering Calculation

We consider the case of discrete and continuous scatterers separately in sections 0and 13.2 respectively. The treatment of energy loss is discussed in section 0.

13.1 Case of a discrete scatterer

With a thin scatterer, one can assume that multiple scattering affects only the track direction, i.e. that it has no effect on its position. The process noise matrix can thus be written as follows

$$Q_k = \frac{\partial(y_k, z_k, \eta_k, C_k, \tan\lambda_k)}{\partial(\theta_1, \theta_2)} \begin{pmatrix} \langle \Theta_1^2 \rangle & 0 \\ 0 & \langle \Theta_1^2 \rangle \end{pmatrix} \left(\frac{\partial(y_k, z_k, \eta_k, C_k, \tan\lambda_k)}{\partial(\theta_1, \theta_2)} \right)^T$$

Equation 13-1

where q1, q2 are uncorrelated scattering angles in two perpendicular planes crossed along the momentum direction. $<Q_1^2>$ and $<Q_2^2>$ are mean squared scattering angles. For small One assumes Given that one assume $<Q_1^2> = <Q_2^2> = <Q^2>$ calculated as shown below

$$\langle \Theta^2 \rangle = \left(\frac{14.1}{p\beta} \right)^2 \frac{X}{X_o}$$

Equation 13-2

One gets after simple algebra

$$Q_k = \langle \Theta^2 \rangle J_k \begin{pmatrix} 1 & 0 \\ 0 & \cos^{-2}\lambda_k \end{pmatrix} J_k^T = \left(\frac{14.1}{p\beta} \right)^2 \frac{X_k}{X_o} S_k$$

Equation 13-3

where X_k is the kth scatterer thickness, $l_k = \tan^{-1}(p_t/p_z)$ i.e. angle between the track projection on the xy plane and the x-axis, and the matrix J_k is given by

$$J_k = \frac{\partial(y_k, z_k, \eta_k, C_k, \tan\lambda_k)}{\partial(\theta_1, \theta_2)}$$

Equation 13-4

13.2 Case of a continuous scatterer

For an infinitely thin scatterer, one can define a differential dQ as

$$\{dQ_k\}_{i,j} = \left(\frac{14.1}{p\beta} \right)^2 \frac{dX_k}{X_o} \{S_k\}_{i,j}$$

Equation 13-5

One can thus calculate the covariance matrix for a continuous scatterer as

$$\{Q\}_{i,j} = \left(\frac{14.1}{p\beta}\right)^2 \int \frac{dX_k}{X_o}\{S\}_{i,j} = \left(\frac{14.1}{p\beta}\right)^2 \int \{S\}_{i,j}\frac{d(X_k/X_o)}{dr}dr$$

Equation 13-6

13.3 Treatment of Energy Losses

The Bethe-Bloch formula expressed for pions, as shown below, enables a determination of the energy loss step by step.

$$\Delta E = \frac{0.153}{\beta^2}\left(\ln\frac{5940\beta^2}{1-\beta^2} - \beta^2\right)\Delta X$$

Equation 13-7

One can thus update the curvature of the track accordingly with the following expression

$$C' = C\left(1-\frac{\Delta C}{C}\right) = C\left(1-\frac{E\Delta E}{p^2}\right)$$

Equation 13-8

14 Appendix 3: Charge of the Integrated Tracker Task Force

The original charge of the Integrated Task Force is included in the inset below.

Monday Nov 13th, 2000
Dear STAR Collaborators,
In the coming year, as we all know, STAR will be upgraded with several additional subsystems to enhance, among other things, tracking precision and efficiency of particle identification. This new setup calls for additional tracking software. While the current approach of tracking in the TPC-only was found to be adequate in the recent review meeting at LBL, and, indeed, has proven to be tremendously successful in the processing and analysis of this year's data, the same review recommended the formation of an R&D group to start work on an integrated tracking scheme to be used for the future detector configuration. The aim of such an approach would be to maximize the tracking efficiency and, at the same time, to minimize the background by 'simultaneously ' using the information of all devices which can measure space-points on particle trajectories. This calls for the investigation into new algorithms for road-finding and track extrapolation, to name only the two main topics. I therefore wish to announce the formation of the Integrated-Tracking Task Force, which will be charged with the following task: To design, test, evaluate, implement, and document an integrated tracker for STAR that:
provides highly efficient and minimum-contamination information on particles emitted into the STAR acceptance.
incorporates all tracking detectors taking into account their detailed geometry, calibration, material location and thickness, as well as magnetic field effects.

provide also tools which allow to extrapolate from one position on the track to any other position along the flight path of the particle with high accuracy (track extrapolation).
Members of this team are chosen based upon their experience, knowledge, and overall skills in computer programming, as well as on their profound understanding of the physics of STAR.
Claude Pruneau from Wayne State University is going to lead this effort.
The group will consist of approximately 10 active members which have to be able to commit a considerable fraction of their time (>40%) to the project. STAR management will assist in configuring the group and in finding the necessary manpower to accomplish the goals.
The following points should be kept in mind:

It is recommended that the group does not initially focus on one approach but investigates and evaluates several techniques for pattern recognition, fitting, and propagation before making a final decision.

Several experiments have worked on this subject in the recent past. Solutions have been developed and implemented by ALICE, Atlas and BaBar, to name but a few. The group should consider the re-use of such existing code or algorithms were applicable and justified.

If, however, a more coherent approach requires the replacement of existing code they should feel free to do so.

The group should stay in close contact with ongoing efforts in the analysis and reconstruction of STAR data and address apparent issues in their algorithms.

The implementation of an integrated tracker requires a sufficiently detailed description of the detector to evaluate energy loss and multiple scattering. It is generally considered that GEANT is too complex, slow, and too detailed to be usable for this purpose. The development of an independent slim geometry interface was identified as one topic of close collaboration between the ALICE collaboration and STAR. The group should pursue this joint effort and take a leading role in its development.

The task force leader should stay in close contact to the STAR reconstruction leader. They both report to the STAR computing leader.

All code is to be written in C++, where necessary compatible with the existing STAR infrastructure.

The groups should meet regularly in phone meetings and at least once per month in person at BNL.