STAR Integrated Tracker Guide

INTEGRATED TRACKER TASK FORCE

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 Section 1. 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 Section 3 of this document. The implementation of the conceptual model is described in Section 4.

1.1 About the Task Force

1.1.1 Task Force Charge

The integrated tracker task force was formed by the STAR computing leader 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.

1.1.2 Task Force Members

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) have recently joined the task force, 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.

We thank Matthias Messer (BNL) who acted as coordinator for the integration of the tracker into STAR production code. We acknowledge the very helpful assistance of Helen Caines (Ohio State/Yale) 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).

1.2 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
- Develop an efficient ghost track rejector and merger.
- 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.

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

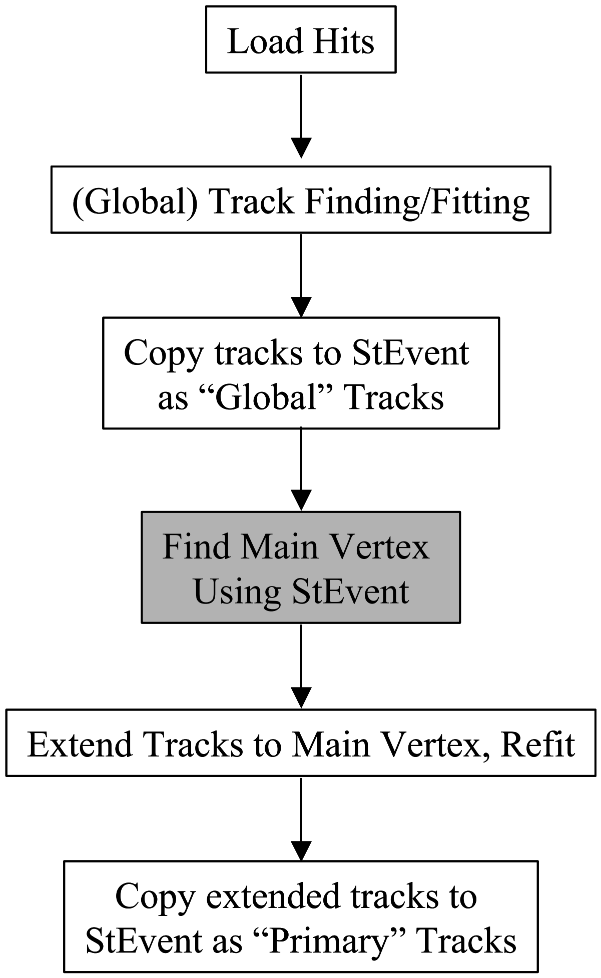

Load Hits
(Global) Track Finding/Fitting
Copy tracks to StEvent as "Global" Tracks
Find Main Vertex Using StEvent
Extend Tracks to Main Vertex, Refit
Copy extended tracks to StEvent as "Primary" Tracks

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.

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.

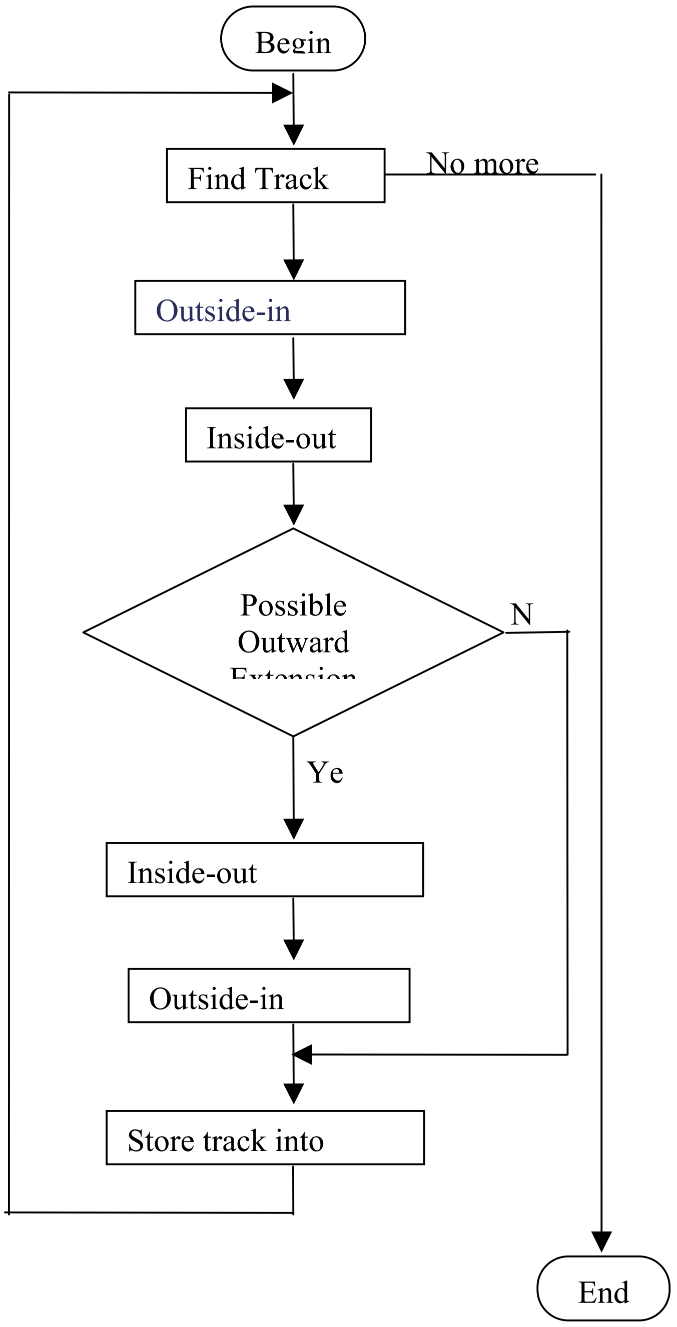


Figure 3-2 Track Search Algorithm

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.

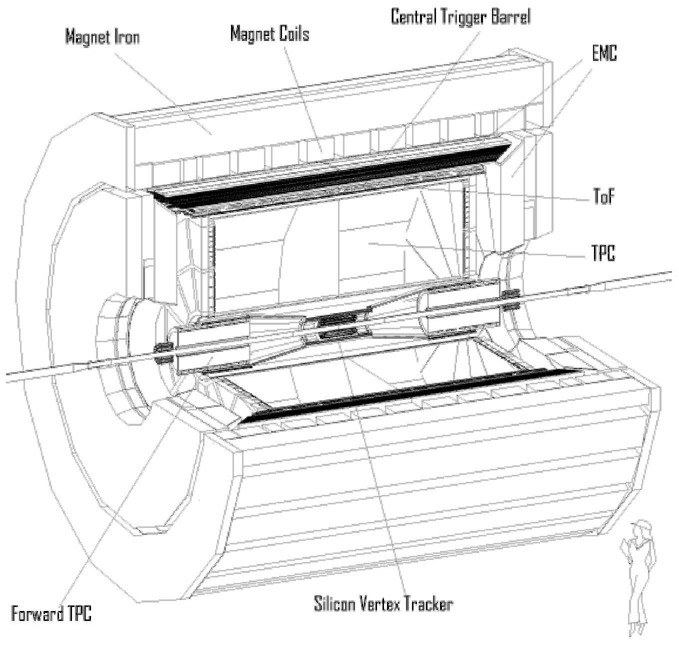


Figure 3-3 Cut away view of the STAR detector

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.

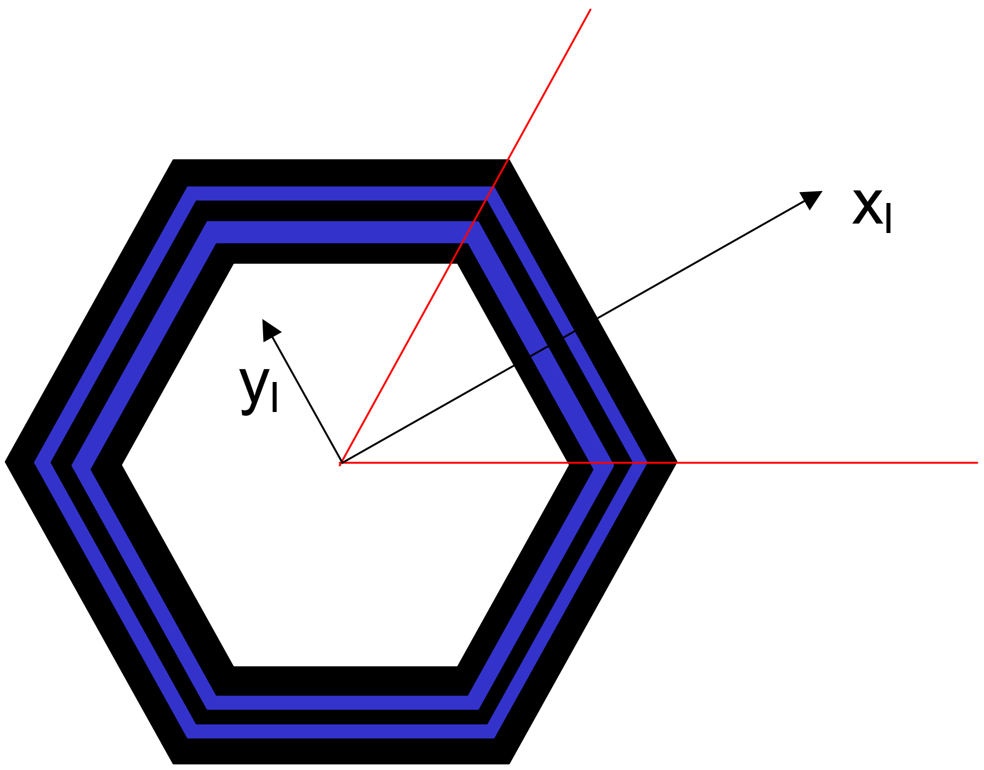


Figure 3-4 Schematic representation of the virtual layers and layer segmentation of a fictitious six-sector detector.

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.

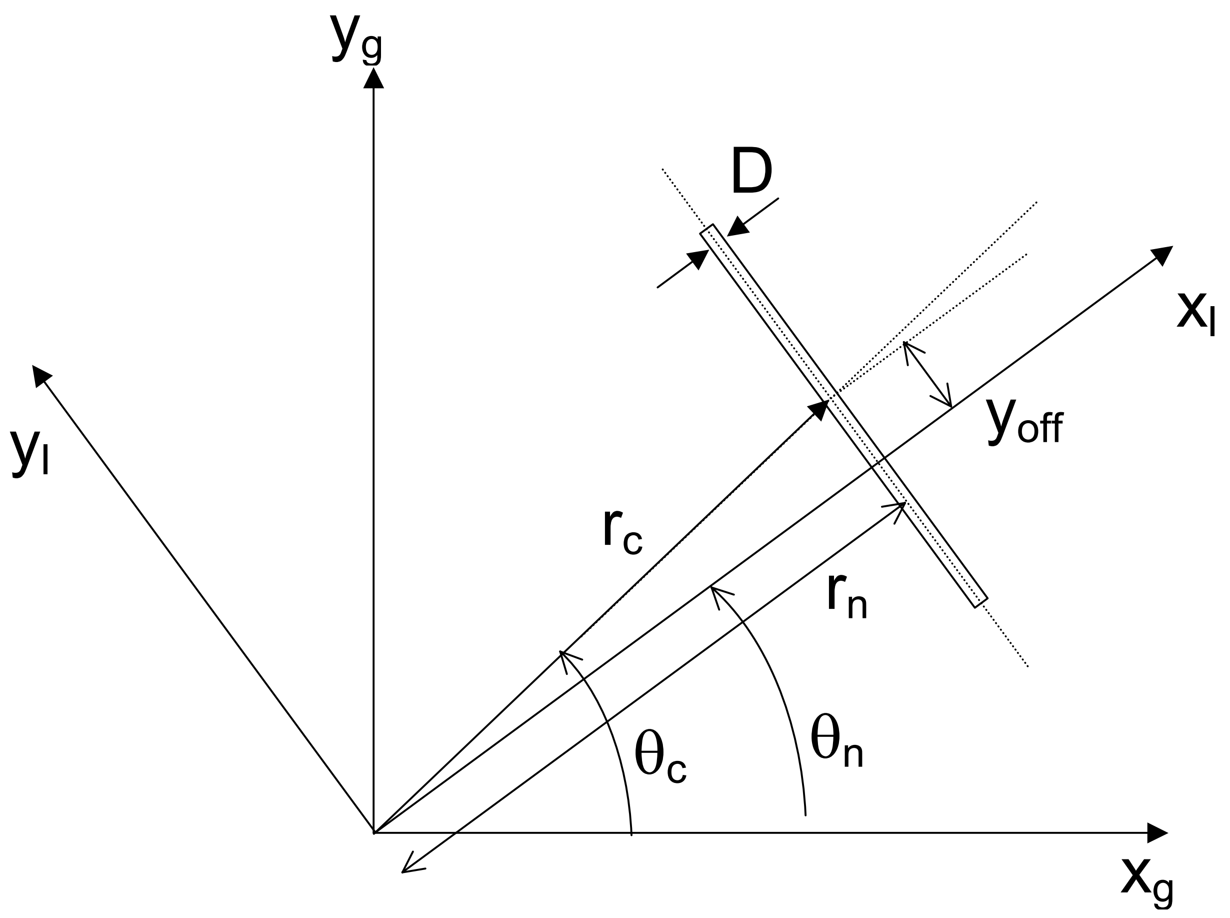


Figure 3-5 Detector Placement

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.

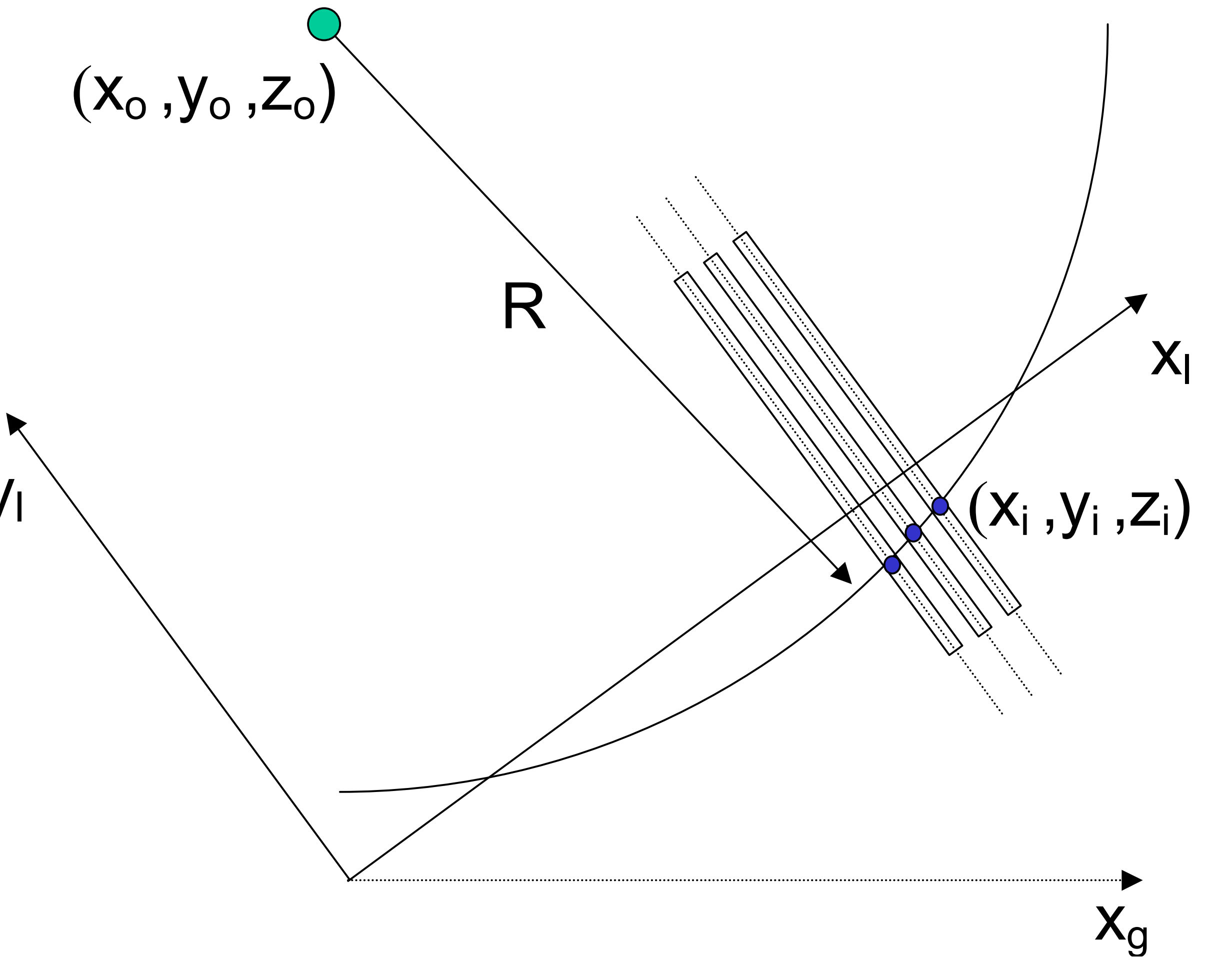


Figure 3-6 Local Track Model

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

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

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:

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

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, η, and tanλ 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:

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

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:

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

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.

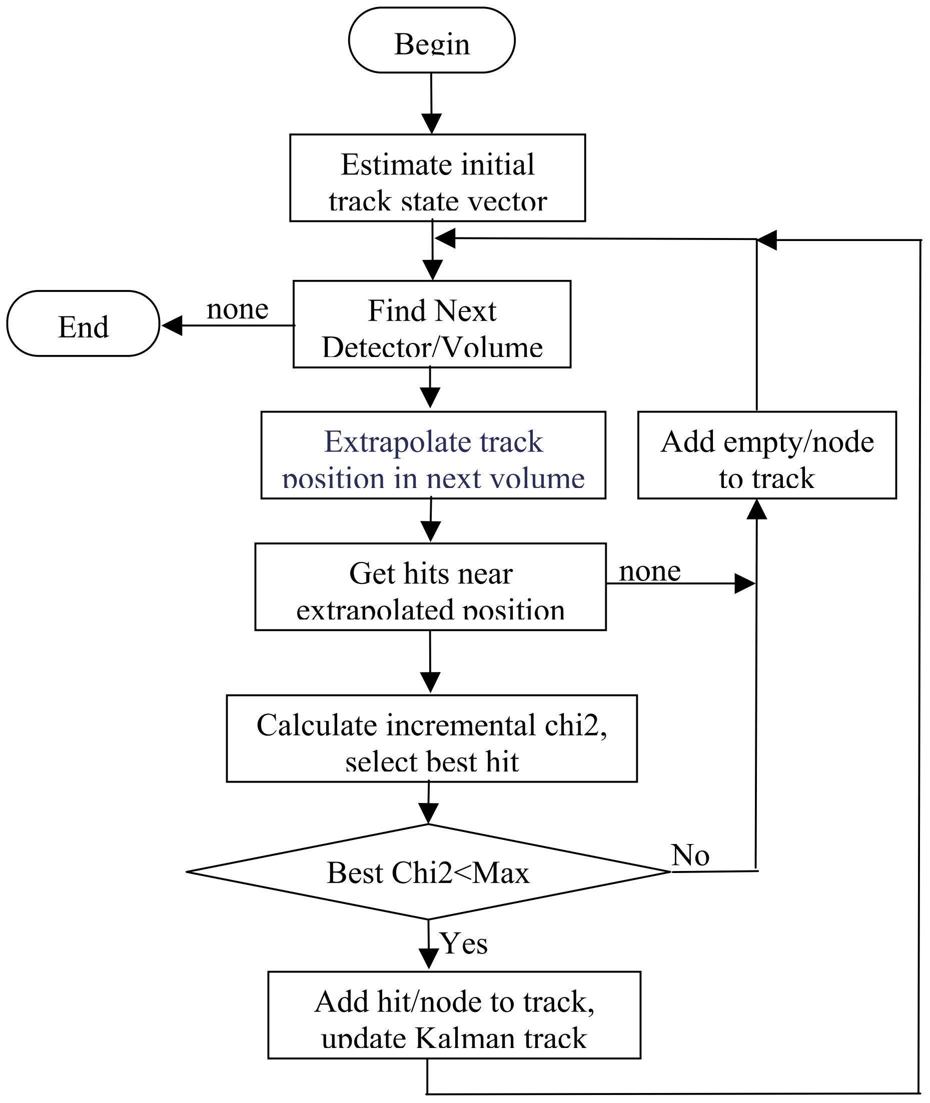

Begin
Estimate initial track state vector
Find Next Detector/Volume
none
End
Extrapolate track position in next volume
Add empty/node to track
Get hits near extrapolated position
none
Calculate incremental chi2, select best hit
Best Chi2<Max
No
Yes
Add hit/node to track, update Kalman track

Figure 3-7 Kalman Search and Fit Algorithm

4 Implementation

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

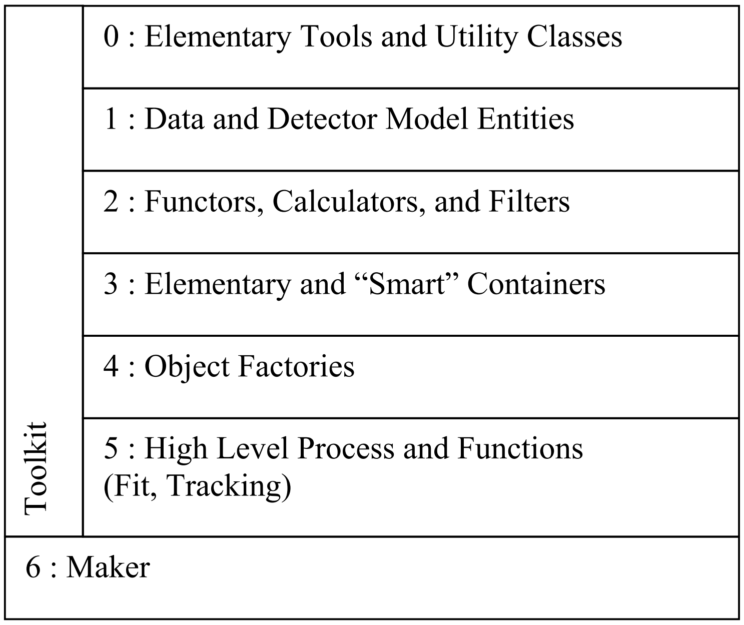
This package was developed during t

he 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 3.2. 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. These are discussed in Section 4.3.5. 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 4.3.6. 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 4.3.7 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 4.3.7.1.



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.

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.

4.3.2.2 SUBJECT/OBSERVER MODEL

A project of the magnitude of the ITTF package has to address the following question, “How do we control the entrance and dissemination of information (e.g., run-time parameters) to a software library of such a large size?” We addressed the issue by first creating a single point of user interaction and second by enforcing a strict pattern for object communication. Thus, all run-time parameters are controlled by a single party, StiIOBroker. By various implemenation strategies, these parameters can be entered either interactively in a ROOT macro or picked up dynamically from a database, depending on a users choice. When new information is passed to the IOBroker, it guaruntees that objects that depend on that information are properly notified.

The communication pattern is an implementation of the Subject/Observer pattern, one of several standard methods for object communication. The Subject/Observer pattern defines a one-to-many dependency between a single subject and many observers. A subject is responsible for notifying its observers when its state has changed. An observer is responsible for requesting specific information from the subject when the subject notifies the observer. This type of pattern is used to implement the relationship between a cell in a spreadsheet and the many charts, graphs, and calculations that depend on the information in that cell. The pattern is implemented in two classes: Subject and Observer. Multiple-inheritance is used to couple an object with one (or more) of these classes. For instance, the class StiIOBroker inherits from the Subject class, and many classes such as StiTrackFinder and StiSeedFinder inherit from the Object class. Then, when run-time information is passed into StiIOBroker, it notifies all of its observers and they call back to get the necessary information that has changed. This is an extremely simple pattern that allows for complicated communication patterns, as any object can be both a subject and an observer. Re-use of the pattern has proved extremely easy, thus justifying the choice of the Subject/Observer pattern over other possibilities such as the "Token" or "Memento" patterns.

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. Simplified class diagrams of these data entities are shown in Fig. X through x'. They form the first tier of the components developed in our tracker.

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.

4.3.3.2 HIT MODELING

The hit modeling required for track reconstruction purposes, as described in section 3.2.5, 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.

4.3.3.3 TRACK MODELING

The modeling of tracks involves a number of classes represented in Fig.

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 3.2.6), 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 3.2.6 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). The list of such specialty classes are presented in table X and Y. A number of mundane tasks related to geometry transforms and material interactions have also been implemented as static methods of the StiGeometryTransform and StiMaterialInteraction classes. These form the second tier components of the system. They are discussed in Section 2b of this document and online as part of the DOXYGEN auto-generated documentation.

4.3.4 Functors, Calculators, and Filters

4.3.4.1 TRACK FILTERS

Track produced by the seed finder and the Kalman Track finder must be filtered to eliminate low quality or irrelevant tracks. Filtering is also used with the event display to select the characteristics of tracks to be shown and for comparative analysis of reconstructed and Monte Carlo tracks. 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.

StiDynamicTrackFilter :

The class StiDynamicTrackFilter is design as a composite filter, a filter consisting of a list of elementary filters. An instance of StiDynamicTrackFilter is initially a null filter. User code can however be used to add elementary filter instances to the container. As filters are added the composite filter becomes increasingly selective. StiDynamicTrackFilter implements the observer/subscriber pattern for object communication. Thus, any change in StiIOBroker is properly propagated into the filtering mechanism.

StiSimpleTrackFilter :

The class StiSimpleTrackFilter inherits from the StiTrackFilter base class as well as the Parameters class. The Parameters class features a "Parameter" container, and accessor to iterators enabling navigation through the container. The Parameter instances can thus be iterated through and used as a basis for making decisions on the acceptability of the track.

4.3.4.2 PARTICLE IDENTIFICATION

One of the primary methods of identifying charged particles in the STAR experiment is through the signature of energy deposition in the detector medium per unit length (dE/dx), using the Bethe-Bloch relation. The particle type of the track is then established as the particle type that yields the theoretical dE/dx value closest to the experimental value. The full calculation of this information from data is a detailed and difficult process; however, using basic approximations the dE/dx can be estimated at the tracking stage.

The most accurate method of calculating the dE/dx uses energy loss corrected for interactions in non-active media and path length calculated through a helix extrapolation. The energy deposited in each pad of the detector without any correction is immediately known, so a dE/dx can be calculated for each layer of the detector in question, and a truncated mean of the values is taken as the first approximation.

The path length of the track in the detector element cannot be established from the hit information itself; it requires the fully reconstructed track. Once a track is reconstructed, in principle the path length is also known explicitly. It is sufficient, however, to make a straight-line approximation to the track for a first order calculation, as illustrated in Figure 4-3.

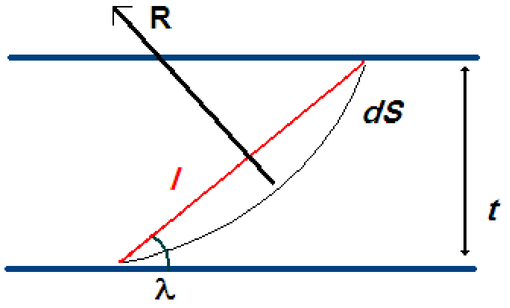


Figure 4-3 dE/dx Calculation

The linear approximation (*l*) to the curved path of a particle in the detector is currently used to approximate the path length. This however will be easily fixed after the review. The detector element thickness, t, and track dip angle λ, are all that is required.

$$dE = E_{deposited}$$

$$l = t\sqrt{(1+\tan^{-2}(\lambda))}$$

Equation 4-1

Figure 4-4 illustrates the results of the calculation.

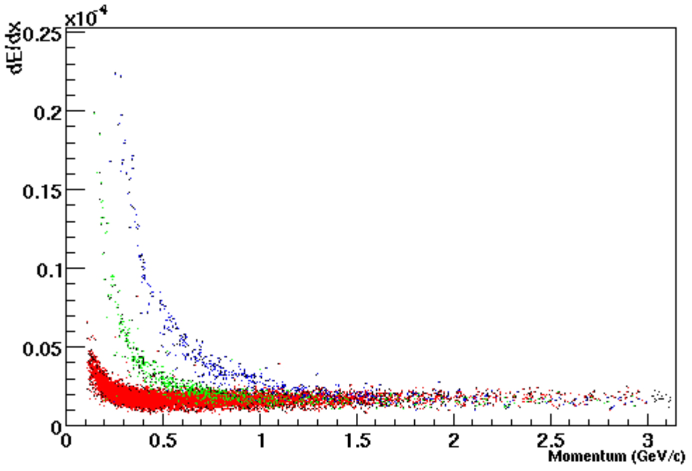


Figure 4-4: Reconstructed Monte Carlo track dE/dx versus momentum. Protons and anti-protons are shown in blue, charged Kaons in green, and charged pions in red.

4.3.5 Containers and Organizers

We make proficient use of STL containers, as they are very well designed, and very efficient in the handling of very large data sets. Yet, STL containers lack the *specific* semantic value needed in handling hits and tracks. We have thus develop classes, which use STL containers “vector” and “map” to hold references and organize data objects, such as hits, tracks, and detector elements. Examples of such containers include the classes StiHitContainer, StiTrackContainer, and StiDetectorContainer. The class StiHitContainer stores all measured hits in hierarchical fashion and enables very fast retrieval of specific hits on the basis, for instance, of their proximity to a given point projected as the extrapolated position of a track in a detector volume. The StiTrackContainer was developed to provide hierarchical storage and retrieval of tracks useful in performance analysis – or potentially also in two track analyses such as HBT or other correlation analyses. The StiDetectorContainer also provide hierachical storage and retrieval of detector and is used in the propagation of tracks through the various detector elements. These 3rd tier components are described online as part of the DOXYGEN auto-generated documentation.

4.3.5.1 DETECTOR MODEL IMPLEMENTATION

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 (~5) different design. 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 classes: StiDetectorContainer and StiHitContainer. These two classes are separate but maintain a well-defined method of communication, so that one can work in both physical space and hit-space simultaneously without introducing a strong coupling between the detector model, the hit model, and the track model. This preserves the charge of writing software that easily integrates information from existing subsystems and those to come in future upgrades.

In the most revised version (stable for more than 5 months) the detector model consists of many instances of detector objects (StiDetector) that represent the physical location of a certain type of material. These objects contain the necessary information to calculate multiple-scattering and energy-loss corrections. For instance, we represent a given TPC padrow and sector (e.g. padrow 10, sector 9) by an instance of StiDetector. Thus, the TPC is represented by 45*12 StiDetector Objects (we treat the east and west sides as one detector. This is not a priori necessary but it is a strong optimization). Other detector elements and scattering centers are treated as well.

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.

Historically, propagating a track through a detector model is a time consuming process. Much effort went into the optimization of the detector container and the results are fantastic. Results are based on the following test:

- Randomly choose an outer layer of the TPC (sector and padrow). Begin swimming inward from this element
- Swim inward from this layer, randomly choosing whether to move in azimuth at each step inward
- Repeat for 10,000 tracks

For the first version of the detector container this process took ~20 cpu seconds. After much profiling and revision, the current version accomplishes the same process in 0.17 cpu seconds – a tremendous improvement!

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.

4.3.5.2 HIT CONTAINER IMPLEMENTATION

Class StiHitContainer is a container designed to handle a very large number of hit instances (of StiHit class), and provide for efficient access to hit subsets within user specified volumes. It is implemented using a mapping between a two dimensional key and an STL container of hits. The mapping is discussed in further detail below.

There is, in principle, a natural connection between the hits and the detector where they were measured. However, for implementation purposes, it is convenient to keep these two entities (StiHitContainer and StiDetectorContainer) separate, while maintaining 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 that 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 the east and west sectors mark a clear distinction in data taking, but that distinction is unimportant in pattern recognition. For more on the nature of the mapping see StiGeometryTransform.

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 >. Class HitMapKey is a simple struct that stores two values: refAngle and position, and MapKeyLessThan is a simple struct that defines a strict less-than ordering for HitMapKey objects. These values are described in the class StiHit. By specifying a HitMapKey, then, one can achieve *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, 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, for instance, one is interested in hits belonging to TPC sector 12, padrow 13. Then, this sector/padrow combination can be easily mapped to a position and refAngle (see StiGeometryTransform). 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 (y_0,z_0) and bounded by +-deltaD in y and +-deltaZ in z. Then, 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(). Alternatively, one can use another method of setRefPoint() that combines all of these steps in one call.

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 filling is performed via class StiHitFiller, which retrieves hits from StEvent and places them in the hit container. StiHitContainer does not own the hits that it stores.

4.3.5.3 TRACK CONTAINER

After a track is found, extended, and accepted by the track filter it is put in a track container (StiTrackContainer). The track container is more than a simple container. It is designed to provide access to a subset of tracks similar in some dimension (e.g., curvature, dip-angle, etc.). This design was chosen to ease future implementation of second pass tracking algorithms that would, e.g., combine two segments of an artificially split track (merging). There are O(1000) tracks per event, and track-track comparisons are therefore an O(1E6) operation – truly intractable. However, by choosing a subset of the tracks with similar characteristics, one can easily reduce this to an O(100) process which is easily accomplished within the cpu limits. The current track container implements this sorting using a STL map, but a beta version is being tested which makes use of a templated home-brew hash container based on STL implementations.

We focus here on the hash table implementation, as it will soon replace the current version. The class StiTrackContainer contains an instance of the templated class StiHash<BinKey, TrackVec, BinKeyLessThan, TrackBinner>. That is, one uses the StiHash class which depends on a key, a container type, an ordering function for the keys, and a hash function. StiHash is just like a std::map with the addition of a functor that takes care of the hashing. Thus, one can define a hash function that bins tracks of similar characteristics. To change the way tracks are stored and retrieved (to increase efficiency) one need not change the container, merely change the hash function. Additionally, class StiHash provides full bi-directional compliant STL iterators. Thus, one can quickly separate the ~1000 tracks in an event into groups of ~10 tracks that have similar characteristics (5 parameters for a helix model, and several other cuts based on number of hits, quality of the fit, etc). Initial timing estimates show the beta version of the hashed implementation to be significantly faster than the current implementation based on a simple STL map, and well within the timing constraints of the project. We plan to roll out the beta version immediately after the current ITTF review. At this point, we can easily begin to treat any problems (if they exist) with track splitting.

4.3.6 Factories, Memory Allocation, and Object Storage

The analysis and reconstruction of events requires a varying number of objects that are instances of hits, tracks, track nodes, and other similar classes, ranging from a few tens in peripheral collisions, to many thousands in central Au + Au collisions. This is obviously not a problem given that C++ enables dynamic memory allocation and release through the use of operators "new" and "delete". It is then possible to allocate, on the fly, as many instances of these classes as needed in a given event – provided of course, one is not exceeding the boundaries of the virtual memory allocated on one's computer… The constant allocation and deletion of memory needed for "small" objects is however a CPU time onerous task. It is thus wise to avoid repeated multiple calls to the class constructor and destructors of these small classes.

Many workaround this problem exist. We have opted to use a factory/container approach whereby objects are allocated and constructed ONCE, kept in a dynamic size container (STL vectors actually) and served through a factory pattern. Served objects are considered reserved. The factories are initialized with a specified number of object instances when constructed. If requests for objects exceed the initial allocation, a specified incremental number of instances are created and added to the internal container. Incremental additions can be repeated. One uses the convention that the objects are never to be allocated directly by "user code" but must instead be requested through their respective factory. The factories feature a "reset" method one calls after the analysis of an event is completed to declare the objects are released and ready to be used for the analysis of the next event thereby freeing the existing objects without "destroying" them. The analysis of successive events may then be conducted without further and repeated memory allocation.

We create and distribute objects using the combination of the "factory" and "memory pool" patterns. All objects are created and owned by a certain object factory. When a new object is needed (e.g., the track finder needs a new track), the user requests a new object from one of several factories. The factory's job is to ensure that the object exists, to manage the memory allocated to that object, and to ensure that new objects are created in a timely manner. The factory owns the objects, thereby explicitly solving the problem of object ownership. Thus, nearly all calls to allocate or destroy memory for objects exist within the factory class, eliminating the problem of memory leaks. Because the project is transient, we can then "reset" the factories after each event and re-use the same objects, further increasing the time efficiency of the code. The prototype code shows a factor of greater than ten in speed compared to standard memory pool implementations. Further, rigorous testing shows that the factory implementation leaks no memory. The factory/pool pattern also offers the obvious advantage that it is possible to decide, at run-time, which factory to use and thereby allow for a choice of simple base type objects, evaluable objects, or drawable objects for use respectively in production, evaluation, or GUI/event display modes.

Object factories are based on the templated Sti class StiObjectFactory<class Factorized>. We have implemented derived classes of this abstract base class for essentially all data and geometry entities used in this tracker. That includes for instance StiHit, StiTrack (and its derived classes), StiTrackNode, StiDetector, StiTrackFilter, Parameter, etc, etc.

The factory model consists of several user defined classes and depends heavily on use of the Standard Template Library (STL). There is a three level lineage to the pattern given below:

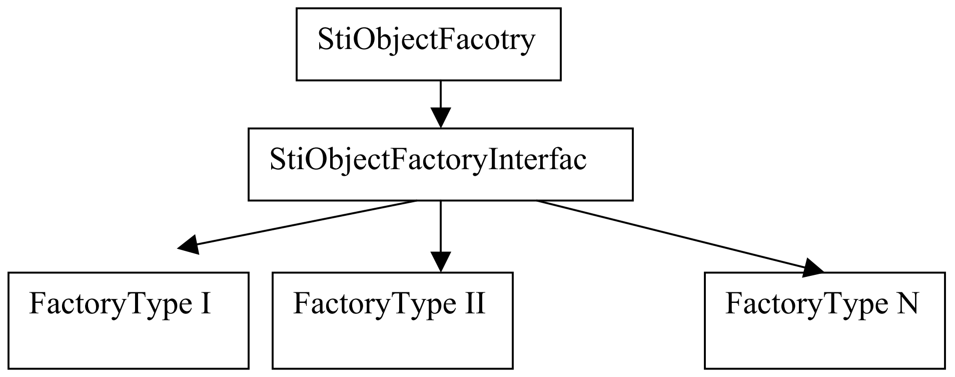


Figure 4-5 Object Factory Conceptual Class Diagram

This structure was designed for the following reasons. First, all object management and destruction is controlled by the StiObjectFactory base class. Specifically, it contains a container of pointers to void (std::vector<void*>) and necessary run-time parameters to control the amount of memory that the factory is allowed to manage. The derived class StiObjectFactoryInterface is templated. Thus, all user interaction with the factories is interaction with an object of type StiObjectFactoryInterface<T>. Templating the class allows for a single implementation of object creation, destruction, and management. Thus, there is only one block of code to maintain. We avoid the common problem of code-bloat (a compiler related issue in C++ use of templates) by the standard method of storing pointers to void with appropriate casting. StiObjectFactoryInterface enforces a strict interface as well as polymorphic use of object factories. For instance, we make a run-time choice to run in graphical or non-graphical mode. But, the tracker itself doesn't care about this decision, because it gets objects from an instance of StiObjectFactorInterface<StiTrack*>. If we run in graphical mode, we simply create an instance of StiObjectFactoryInterface<StiDrawableTrack*> instead. Since StiDrawableTrack derives from StiTrack, these two can be used interchangeably. Thus, the difference between graphical and non-graphical running consists of one line of code – deciding which type of factory to create. All such decisions are managed by the StiToolKit class. Finally, at some stage one has to actually know what types of objects to create. This is the reason for the derived classes (above represented by Factory Type I, etc). These are skeleton classes whose only job is to implement the function (void* makeNewObject()). This function makes the appropriate call to operator new. All calls to makeNewObject() happen in the base class StiObjectFactory. All derived factory types are contained in a single file StiFactoryTypes.

Further information about these 4th tier classes can be found online as part of the DOXYGEN auto-generated documentation.

4.3.7 High Level Processes

Track seed finding, track finding and fitting, are complex operations or processes, and are, as such, handled by complex classes making use of a large number of subsidiary classes. We have designed the tracker components based on the notion that track seed finding, track search, and track fitting can be considered distinct tasks or operations. They are described in the next three sub-sections.

4.3.7.1 TRACK SEED FINDER

The seed finding is implemented following the hierarchy illustrated in Figure 4-6.

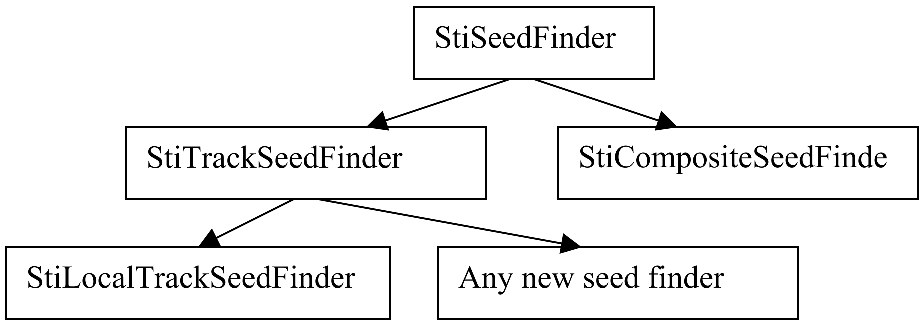


Figure 4-6 Track Seed Finder Class Diagram

StiSeedFinder is an abstract base class defining the notion of a seed finder. It provides virtual methods hasMore() and nextTrack() which can be respectively be called to find out whether additional seeds are available and to obtain seeds in sequence. StiSeedFinder is an abstract class and as such only provides an interface to a finder.

StiTrackSeedFinder is also an abstract class but provides so data members (primarily factory pointers) and defines the skeleton of an algorithm that creates StiKalmanTrack objects.

StiLocalTrackSeedFinder implements the road-finder algorithm presented in Section 3.2.3. Its implementation is described in some details below.

StiCompositeSeedFinder was implemented to allow for multiple types of seed finders to be used at the same time. StiCompositeSeedFinder holds a container (std::vector<StiSeedFinder*>) of seed finders and loops through them in a user defined order. Currently, only the StiLocalTrackSeedFinder algorithm is used and added to the composite finder, but we foresee the addition of a second pass algorithm to find those tracks that are missed by the road-finder. It should be noted that StiCompositeSeedFinder already provides the "plug-and-play" functionality required by the ITTF charge.

The algorithm used by StiLocalTrackSeedFinder can be summarized as follows:

1. Begin at a run-time specified detector element (current version begins at padrow 45, sector 1).
2. For every hit in that padrow (that is not already assigned to a track) extend the seed (see explanation below).
3. When all hits are exhausted in that detector layer, move to the next layer in azimuth (padrow 45, sector 2).
4. Go to step II.
5. When all sectors are exhausted, move in to the next layer (e.g. padrow 44, sector 1).
6. Go to step II.
7. Stop when a stopping point is reached (currently padrow 6).

Track seeds are extended as follows: Given a seed point, the hit that is closest in position space in the next layer is chosen as the other half of the two-point seed. If that hit happens to be beyond the tolerances specified, the seed is aborted, and the process is begun again with the next hit in the padrow. Once the two point seed is identified the seed finder uses straight line extrapolation to add hits in each padrow, moving in towards the origin. A seed is allowed to skip one or more layers without having a hit. When the seed reaches a pre-determined stopping point (currently 6 layers) the seed is fitted first in the bend plane with a fast circle fit algorithm (StFastCircleFitter from the StarClassLibrary) and then in the path length vs. z plane with a fast line fitter (StFastLineFitter, currently being added to the StarClassLibrary). The result of the fit is then used to initialize an instance of StiKalmanTrack. It is also possible to use 3 points to calculate the helix parameters, but initial testing showed the results to be less robust than a fit to 4-6 points. It should be noted that the hit errors are not used in the fast fit. This process is iterated for each hit in the padrow.

4.3.7.2 TRACK FINDER

The track finder is implemented following the class diagram presented in Figure 4-7.

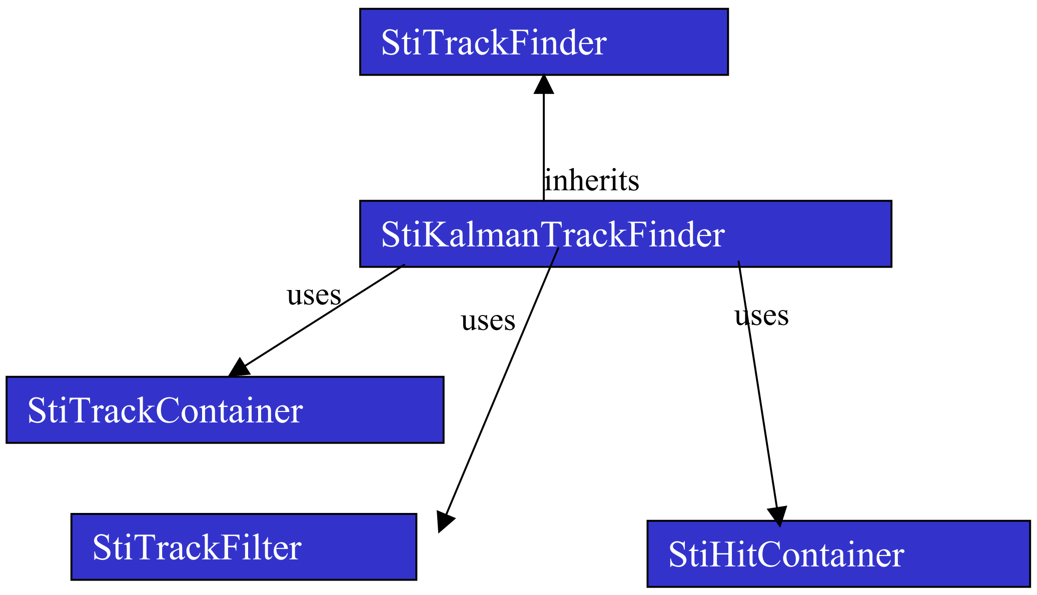


Figure 4-7 Track Finder Class Diagram

4.3.8 Toolkit

The operation of the tracker necessitates the deployment (e.g. instantiation) of a considerable number of entities whose type must be determined at run-time depending on the operational mode (production, evaluation, or event display): evaluable or drawable data and geometry entities must be used, rather than the basic production entities, if used in evaluation or GUI mode. A master factory, or toolkit, is thus used to serve the classes appropriate to the given operation conditions. The toolkit is also used to guarantee classes, which have mutual dependencies, are created in proper sequence, and thus insure correct initialization of the code.

The toolkit is defined as an abstract singleton class. Currently, one derived class, StiDefaultToolkit, exits for operation within root4star, and analysis of Star data.

Note finally that the toolkit pattern deployed also enable flexible detector and geometry conditions: major components are instantiated only if and when needed. The code footprint is thereby minimized and the code initialization reduced optimally.

Further information about the toolkit classes can be found online as part of the DOXYGEN auto-generated documentation.

4.3.9 Event Display

Track finding is a pattern recognition problem—a problem at which the human mind excels. Thus, the only way to truly test, debug, and gain confidence in a tracker is to see it work. The ITTF graphical user interface (GUI) was designed and implemented for this purpose. After initial investigations, it was recognized that STAR computing environment did not provide a graphical tool that allowed a user to interface with c++ objects. Several event displays did exist, but all were made for “picture taking” instead of interaction. A debugging GUI should allow one to perform a function, look at the results, change some parameters, and repeat the exact same function, without exiting the program or restarting. Thus, a GUI was built from scratch that allows one to graphically track events step-by-step, rewind, modify, and repeat. The GUI has proved a powerful debugging tool, but it is not meant as a production version of a STAR event display. The GUI is one part of the project that depends on the ROOT libraries, but it was implemented in such a way that a clear separation of the ITTF project from the ROOT libraries is maintained. We use polymorphism and interfaces to ensure that the GUI could indeed be implemented using any number of graphical libraries (e.g., OpenGL, OpenInventor, QT, TCL, etc…). The choice of ROOT for the specific implementation reflected the fact that ROOT was the only graphical library available in the STAR environment at the time the project began.

The GUI allows one to view the detector model, hits, and tracks in various levels of complexity. It provides a full 3d view of the objects used in the project. Furthermore, it allows for interactive modification of the run-time parameters. One could easily argue that, were it not for the GUI, we would have progressed at a far slower pace, and we would ask the STAR computing leadership to strongly examine the addition of a unified interactive graphical library to the STAR computing environment. We spent months of valuable author time concentrating specifically on the GUI, and it would be a shame for another project to have to dedicate such a large fraction of resources to the same thing.

The Gui is implemented exclusively in the library StiGui. Polymorphism is the general tool. The top level is class StiDisplayManager which holds a container of StiDrawable objects. The Sti library depends only on StiDisplayManager. We derive class StiRootDisplayManager from StiDisplayManager to control the ROOT implementation of the GUI. The display manager operates on objects of type StiDrawable. These are virtual, polymorphic objects that define the interface expected of an object that can be rendered to the screen: color, line type, marker type, visibility, etc. Next, we derive class StiRootDrawableTrack, StiRootDrawableHit, StiRootDrawableDetector, etc. using multiple inheritance. For instance, StiRootDrawableDetector derives from classes StiDetector and StiDrawable. Thus, by simply deriving from class StiDrawable an object magically shows up on screen.

StiDisplayManager defines many methods to control the view, scale, angle, etc. of the canvas. StiDisplayManager is in no means a large scale solution to GUI implementation. More complex graphics (composite objects, motion, etc) require much more complicated structures (ordered trees, etc). However, StiDisplayManager is more than sufficient for the purposes of the ITTF project.

5 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
$\mathbf{x_{k+1}^k}$	the extrapolated state vector from point k to point k+1
$\mathbf{C_{k+1}^k}$	the covariance matrix of $\mathbf{x_{k+1}^k - x_{k+1}^t}$, where $\mathbf{x_{k+1}^t}$ 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:

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

$$\mathbf{m_k = H_k\, x_k + e_k},$$

where $\mathbf{x_0}$, is an initial estimate of the state vector, $\mathbf{F_k}$ is the state propagator matrix at the kth point, $\mathbf{Q_k}$ the process noise covariance matrix evaluated at the kth 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:

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

5. $x_k = F_{k-1}\, x_{k-1} + K_k\, r_k^{k-1}$
6. $C_k = (1 - K_k H_k)\, C_k^{k-1}$
7. $(\chi^2)^k_{k-1} = (r_k^{k-1})^T (R_k^{k-1})^{-1}\, r_k^{k-1}$

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 $\boldsymbol{\chi^2}$ as evaluated in step 7 above.

6 Appendix 2: Multiple Scattering Calculation

We consider the case of discrete and continuous scatterers separately in sections 0and 6.2 respectively. The treatment of energy loss is discussed in section 0.

6.1 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 6-1

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 6-2

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 6-3

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 6-4

6.2 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 6-5

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 6-6

6.3 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 6-7

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 6-8

7 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:

1. provides highly efficient and minimum-contamination information on particles emitted into the STAR acceptance.

2. incorporates all tracking detectors taking into account their detailed geometry, calibration, material location and thickness, as well as magnetic field effects.

3. 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.

8 Some Open Issues

Some open issues are discussed in the section(s) below.

8.1 Seed Finder

The local seed finding algorithm was implemented and is currently being tested. Unfortunately, quantitative comparisons to the current track finder have only recently become possible, so we are learning a great deal about the pattern recognition at the time of the review. Initial results show that the seed finder is quite good at finding tracks that are reasonably straight (pt>600 MeV). However, it appears the current efficiency is too low for tracks of large curvature. The cause of this has not yet been established, but we will begin debugging immediately after the review. Below I list a bulk classification of the issues to be investigated.

- Identification of two-point seeds. This is controlled by two parameters, deltaY and delta Z, which control how close in position space two points must be to be considered a two-point seed. Obviously the dependence on these two parameters (and any bias they might introduce in, e.g., dip-angle) must be studied.
- Extrapolation of two-point seeds. This controlled by two parameters, as well. Further, there is a known bug that has to be fixed that deals with conversion of integer numbers (e.g., pad number or time bucket) to floating point numbers. This has a small but finite effect on efficiency for low momentum tracks.
- Length of seeds. The seed finder stops when a track reaches a certain number of points. Additionally, it allows for a finite gap in the seed. That is, a track can miss a hit in one or more pad rows and continue. These parameters must be varied and their effects studied.
- Helix fit. Once a seed has been found, we perform a circle fit in the bend plane and a straight line fit in the s-z plane. If the fit fails the seed is aborted.