The BaBar Tracking System

S.F. Schaffner
for the BaBar Computing Group

SLAC, P.O. Box 4349, Stanford, CA 94309 USA
sschaff@slac.stanford.edu

An object-oriented system for charged particle track reconstruction developed for use with the BaBar detector is described.

Key words: Object-oriented; tracking; BaBar.

1 Tracking

Historically, charged-particle tracking code is complex, is hard to maintain, and is the focus of constant efforts at improvement. It therefore provides an excellent opportunity to gain some of the potential benefits of object-orientation in the context of high energy physics reconstruction software. This paper describes the design of an object-oriented tracking system for the BaBar detector. Our goal in creating the design is to produce a system that is highly modular, with clear interfaces within tracking and with other components of reconstruction and analysis. In particular, it should permit changes to any part of the system – pattern recognition techniques, fitting algorithms, material model, calibration parameterization, and detector hardware – without requiring redesign of the rest of the system. It should be flexible enough to accommodate special-purpose applications (e.g. a detector alignment package, or reconstruction of field-off cosmic rays), and provide considerable functionality for consumers of tracks (e.g. consumers should be able to refit a track, or calculate the intersection point of a track with a detector element, without having to write their own code). And, of course, it should be fast. The design sketched below is still evolving, but the key features have all been implemented in C++ and appear workable; remaining details will be decided upon as the code moves from prototype to production status.

The central concept in tracking is of course the track. In the BaBar design, a track object (class RecoTrk) describes a path through space and time; it is typically (but not always) associated with a set of detector hits. A RecoTrk accordingly consists of a path-description object (derived from the abstract class TrkRep) and a (possibly empty) list of hits, along with some global information about the track (e.g. its charge, and flags describing its current status). Hits can be added and removed in the course of reconstruction, and one TrkRep can be replaced by another; for example, initial pattern recognition might be done with a simple helix TrkRep, while final fitting will be done with a more complex representation appropriate to a Kalman filter. In order to handle multiple fits of the same track using different mass hypotheses, the RecoTrk can in fact maintain multiple TrkReps simultaneously, one for each hypothesis; alternatively, it can store only the preferred hypothesis for that track, and generate others on demand.

In order to preserve flexibility, no particular parameterization is assumed for the representation of the track's path. Rather, the description is given in terms of an abstract Trajectory class; this defines functions that return, as a function of path length along the trajectory, the particle's position, direction, and vector curvature. It also specifies functions giving the maximum permitted extrapolation (for a given tolerance) for the linear and parabolic approximations. Classes exist for calculating the intersection of one of these Trajectory objects with an arbitrary surface, and for calculating the point of closest approach of two Trajectories. Thus, while a simple helix TrkRep will contain a derived helix Trajectory, all geometric operations are done using the base class interface. (It will be possible, however, to substitute code for particular pairings – e.g. the intersection of a helix with a cylinder – without changing any existing code, should that be needed for efficiency reasons.) All communication between tracking and the detector model is done through Trajectory objects.

The TrkRep exists to interpret the Trajectory it contains. Specifically, it knows what mass hypothesis should be assumed, and can calculate the arrival time of the particle as a function of path length. In addition, it can invoke the appropriate fitter for fitting the Trajectory to an input set of hits.

Given this arrangement, the RecoTrk user interface responds to most requests by getting information from its TrkRep(s), or from the Trajectory stored in the TrkRep. Geometric information, including position, distance of closest approach, and intersection with detector surfaces and volumes, are calculated as a function of path length using the Trajectory momentum is similarly calculated from the curvature of the Trajectory. Information on the particle's time of arrival, and on the quality of the fit, are obtained from the TrkRep. Finally, the TrkRep can supply a standard set of five helix parameters, also as a function of path length, for communication with packages that use more traditional approaches to tracking.

Users of RecoTrks do not have direct access to TrkReps, and cannot directly create RecoTrks. Instead, tracks with a particular representation are created by a factory object (deriving from abstract class FitMaker) of a type appropriate to that representation. FitMaker objects are also used to change the representation of a track; the new TrkRep uses the output of the previous one to provide seed parameters. RecoTrks offer a fit() function, so that all fitting is done through the track interface itself.

The usual starting point for creating tracks is a set of Hit objects, which contain feature-extracted data from the tracking detectors. These are the primary objects that are encountered in pattern recognition. Since most pattern recognition is carried out using algorithms tailored to a particular subdetector, little in the way of a common interface has been required. One exception is a function for calculating the distance between a track's trajectory and a hit.

Hits also contain a record of which tracks they are currently being used in, in the form of a list of HitOnTrack (HOT) objects. The HOTs record the details of the Hit's use on a particular track, and are the objects that the RecoTrk retains in its hit list. They cache information (e.g. residual, path length along track), and store any other information needed to carry out the fit (e.g. the left-right ambiguity choice for drift chamber hits). They also contain flags that permit disabling their inclusion in a fit. They are responsible for calculating their residual, and its derivatives, with respect to the track. Currently, this is implemented by representing the hit itself as a second trajectory, and calculating the distance of closest approach between the two trajectories. The derivatives are calculated automatically, by carrying out the residual calculation using "differential numbers": objects that contain both the value of a quantity and its derivatives with respect to some set of parameters, in this case the parameters of the track Trajectory. In this way, HOTs can perform their calculations without ever knowing what kind of Trajectory is being used to describe the track.

This approach gives great flexibility. Geometric operations are largely divorced from purely tracking operations. Subdetector- specific details are all hidden behind common interfaces. Many different representations can be treated in a uniform way; the user of the track need never know what representation is currently present. Because HOTs carry out much of the actual fitting calculation, fitters tend to be simple. Because operations are done through generic interfaces, a single fitter can be used to fit many kinds of simple representation (e.g. a kinked helix, a spiral, a straight line, a helix with t_0 as a free parameter). Non-standard input to fitting – e.g. the location of the interaction point, hits in a particle id device – can be incorporated as HOTs on the same footing as input from the standard tracking devices; entire track segments, in fact, could be handled the same way.