HPS Kalman Filter Documentation January 7, 2021

R.P. Johnson U.C. Santa Cruz

Contents

1 Introduction

The Kalman Filter code in hps-java (org.hps.recon.tracking.kalman) is intended ultimately to provide a new pattern recognition code that should have the following advantages over the existing code:

- No use of 3-D hits, ever, so strip hits can be picked up and fit even if there was no hit in the other layer of a pair.
- Rigorous treatment of multiple scattering (in the Gaussian approximation) throughout, as track candidates are being followed to pick up hits.
- Rigorous helix model, including field non-uniformity (except in the 3-layer seed that is used to initiate the Kalman Filter).
- Hopefully higher speed.

My approach to developing the code was to minimize outside dependencies so that it could easily be run in a stand-alone mode during development. The stand-alone code builds and runs with extremely fast turn-around, a big advantage in development. It operates on an internal "toy" Monte Carlo simulation that uses fourth-order Runge-Kutta integration and multiple scattering simulated with Gaussian random numbers to propagate tracks through the detector layers. Gaussian random numbers are also used to simulate the detector strip resolution. The advantage of this over using the full-blown Geant-4 simulation is not only that it runs extremely fast, but mainly that it produces tracks that should fit perfectly the fitting model and therefore yield results that can be directly compared with mathematical expectations, to verify that the code is doing exactly what is expected.

Of course, then the code does have to be tested on a realistic simulation, so there are interfaces to hps-java, initially written mostly by Miriam. Miriam's original driver, KalmanDriverHPS.java, was designed to refit existing HPS tracks in order to test the Kalman Filter fitting code. The driver intended for production is KalmanPatRecDriver.java, which runs the pattern recognition in addition to the Kalman Filter.

2 Execution Control

The following cards are available for use in the steering file for driver KalmanPatRecDriver.java:

- logLevel: set the logging level in the driver. Example:

```
<logLevel type="String">CONFIG</logLevel>
```

- siHitsLimit: integer maximum number of hits allowed in the event. Events with a larger number are skipped
- seedStrategy: set a single seed strategy for the pattern recognition in both the top and bottom trackers. The value must be a string of exactly 7 characters, where each character corresponds to a tracker double layer. The character '0' means that the

double layer is not to be used. A 'B' means to use both the axial and stereo layers in the double layer, 'A' means to use just the axial layer, and 'S' means to use just the stereo layer. Lower case letters are also recognized. Each seed strategy must consist of exactly 3 stereo layers and 2 axial layers. There is no limit to the number of seed strategies. An example is

```
<seedStrategy type="String">000BBS0</seedStrategy>.
```

Redundant strategies and invalid strategies get flagged by messages and eliminated.

- seedStrategyTop: set a single seed strategy for the pattern recognition in only the top tracker.
- seedStrategyTop: set a single seed strategy for the pattern recognition in only the bottom tracker.
- numStrategyIter1: in integer specifying the number of seed strategies to be used in the first iteration of the pattern recognition. In this way, some of the strategies can be reserved to be used in just the second iteration.
- doDebugPlots: boolean, true or false, to turn on production of development histograms in KalmanPatRecPlots.java. This uses a lot of extra execution time, so keep it off when not needed.
- numEvtPlots: an integer specifying how many single-event displays to output. Each plot goes into a Gnuplot (.gp) text file and can be displayed by Gnuplot.
- useBeamPositionConditions: boolean, true or false, to turn on use of the beam location from the conditions database.
- useFixedVertexZPosition: boolean, true or false, to force the Z beam position not to be used from the conditions database.
- beamSigmaX, beamSigmaY, beamSigmaZ: floating-point beam-spot size in mm
- beamPositionX, beamPositionY, beamPositionZ: floating-point beam-spot position in mm
- seedCompThr: floating point threshold for comparing seed helicies for redundancy
- addResiduals: boolean, true or false, to add the hit-on-track residuals to the LCIO event
- numPatRecIteration: integer number of iterations of the Kalman pattern recognition (normally 2)
- numKalmanIteration: number of iterations of the Kalman helix fit in the final track fit
- maxPtInverse: floating point maximum value of 1/pt in 1/GeV for seeds and for the final track

- maxD0: floating point maximum d_ρ (or d_0) in mm at the target plane for a seed and the final track
- maxZ0: floating ponit maximum d_z (or z_0) in mm at the target plane for a seed and the final track
- maxChi2: floating point maximum Kalman χ^2 per hit for a track candidate
- minHits: minimum number of hits on a Kalman track
- minStereo: minimum number of stereo hits on a Kalman track
- maxSharedHits: maximum number of hits on a track that are shared with other tracks
- maxTimeRange: floating point maximum time range in ns spanned by all the hits on a track (or a seed)
- maxTanLambda: floating point maximum of —tan(lambda)— for a seed helix
- maxResidual: floating point maximum residual, in units of SSD resolution, to add a hit to a track candidate
- maxChi2Inc: floating point maximum increment in χ^2 to add a hit to an already completed track candidate
- minChi2IncBad: floating point minimum increment in χ^2 to remove a hit from an already completed track candidate
- maxResidShare: floating point maximum residual in units of detector resolution for a shared hit
- maxChi2IncShare: floating point maximum increment in χ^2 for adding a shared hit to a track
- mxChi2Vtx: floating point maximum χ^2 for a 5-hit track with a vertex constraint
- minSeedEnergy: minimum energy of hits for them to be used in seed formation.
- lowPhThresh: improvement in the residual necessary to justify using a low pulse-height hit instead of a high pulse-height one when adding hits to track candidates. The minSeedEnergy parameter is used to distinguish between low versus high pulse height.
- edgeTolerance: tolerance, in mm, for checking whether a track extrapolation is within the detector bounds.

3 Default Parameters

Default parameters and cuts for the Kalman pattern recognition are established by KalmanParams.java, of which a single instance is created by KalmanPatRecDriver.java. Most of the parameters can be changed from the steering file (see Section 2). A complete set of seed strategies is established by default, but if even a single strategy is input from the steering file, then all of the default strategies are deleted. Therefore, if any strategies are introduced from steering, then the whole set of them must be included. All of the parameters, cuts, and seed strategies are printed by the print method of KalmanParams.java, which is called by KalmanPatRecDriver.java at the beginning of the run, so look in the log file to see the current values.

4 Code Organization

The code all lives in the package org.hps.recon.tracking.kalman. The interface to the HPS-Java world is the class KalmanInterface.java. There are two driver classes: KalmanDriverHPS.java and KalmanPatRecDriver.java. The former was primarily developed to test the Kalman-Filter by refitting existing GBL tracks, using the hits already associated by the existing HPS tracking code. It and the associated fit driver KalmanTrackFit2.java have mostly served their purpose and are no longer used or maintained. KalmanPatRecDriver is the driver intended for use in data reconstruction, as it starts with the full set of reconstructed hits and performs pattern recognition as well as fitting. Associated with KalmanPatRecDriver is the class KalmanPatRecPlots, which is used to produce histograms and event displays for development and debugging. The class Efficiency.java is used by KalmanPatRecPlots to make some efficiency plots for development work.

The pattern recognition is based on the following classes:

- KalmanPatRecHPS.java, the main routine for the pattern-recognition flow. It includes a method for driving the Kalman filter, which is used to propagate each track candidate and pick up hits.
- TrackCandidate.java, which provides the data and methods for complete Kalman-Filter track candidates, including a method to refit a candidate.
- KalTrack.java, into which the final, accepted track candidates are accumulated. It also includes a method to refit the track, which in fact always provides the final track fit.
- KalHit.java, used to keep track of the available hits per tracking layer.
- KalmanParams.java, encapsulates all the parameters used to control the pattern recognition. The driver program KalmanPatRecDriver makes it possible to set most of these from the steering file.

One class is used to provide initial helix parameters for the Kalman Filter:

- SeedTrack.java, which drives the linear fit and then converts the results into helix parameters, including covariance matrix. The LinearHelixFit method fits a set of hits simultaneously to a parabola (in the bending plane) and a line (in the non-bending plane). The hits are assumed to be from strip measurements on planar detectors. The

parabola and line fits have to be simultaneous (*i.e.* a 5-parameter fit) in order to account for the information from stereo layers.

- LinearHelixFit.java is a deprecated class. It's code was subsumed into the SeedTrack class in the conversion to using the EJML matrix package, to improve efficiency.

Besides providing the helix parameters needed to start the Kalman Filter, this set of code also is heavily used in the first stage of the pattern recognition to test pattern-recognition seeds.

The Kalman Filter track fitting itself is encapsulated in the following classes:

- Measurement.java, to hold the individual hit information for access by the fitting (and pattern recognition) code.
- SiModule.java, one instance for each silicon-strip module. It holds the list of hits (measurement.java) plus geometry information and methods.
- MeasurementSite.java, one instance for each silicon-strip module that provides a hit to a given track. A track fit is an array of these MeasurementSites. Methods in this class implement the Kalman Filter, together with the methods in StateVector.java.
- StateVector.java, describes the Kalman-Filter helix state vector (five helix parameters) and the methods needed to propagate them, filter them, and smooth them.
- HelixState.hava, describes a local helix. This is more than just a set of five helix parameters and covariance, see Section 7. The local coordinate system is defined within, so that the helix parameters can be defined in a non-uniform field.
- HelixPlaneIntersect.java, provides the math to calculate where the helix intersects a plane. This includes a numerical integration option as well as an analytic helix option, see Section 8.

Then there are several classes the provide mathematical support. Of course external libraries could be used for these, and in some cases are, but they are trivial bits of code for the most part, which originally made it easy to do stand-alone development. They are used only within this package, and I see no point in changing them now in favor of rewriting a lot of code to use external libraries.

- Vec.java, n-vectors, where typically n is 3 or 5 (for helix parameters in the latter case).
- SquareMatrix.java, typically used to implement 5×5 covariance matrices. This has been largely superseded by use of the EJML package (Efficient Java Matrix Library), making use of the procedural calls, instead of object-oriented coding, to maximize speed (the object-oriented approach results in too much copying and recopying of matrices from one memory location to another).
- RotMatrix.java, 3×3 rotation matrices, typically for coordinate transformations.
- RungeKutta4.java, propagation of a charged particle in a magnetic field, by numerical integration. It is very simple, with a fixed step size (*i.e.* not adaptive).
- Plane.java, description of a simple plane in terms of a point and direction.

There are also a number of classes that are in the package solely for the purpose of testing and debugging in a stand-alone mode, outside of HPS-Java, making use of a built-in idealized track simulation. None of these get executed within HPS-Java. They are

- Helix.java and RKhelix.java, used to simulated tracks, including Gaussian measurement errors and Gaussian multiple scattering.
- HelixTest3.java, used to test the Kalman-Filter track fitting mathematics and coding using idealized measurements and scattering.
- FieldMap.java, which provides an interface to the HPS field map in the stand-alone simulation, using a local copy.
- Histogram.java, used to make stand-alone histograms, which can be displayed using Gnuplot.
- PatRecTest.java, used to test the pattern recognition using the idealized stand-alone simulation.
- TestMain.java, the top-level program for the stand-along simulation.

5 Helix Parameters

The Kalman code uses the same set of helix parameters as the KalTest package (https://www-jlc.kek.jp/subg/offl/kaltest/). The parameterization assumes that the magnetic field is aligned with the z axis, and the parameters depend on an arbitrary choice of "pivot point" $\vec{x}_0 = \{x_0, y_0, z_0\}$. They are $a = \{d_\rho, \phi_0, \kappa, d_z, \tan\lambda\}$, where d_ρ is the distance of the helix from the pivot in the x, y plane, ϕ_0 is the angle from the x axis to the pivot, rotating about the helix center, $\kappa = Q/p_t$ is the charge-signed reciprocal transverse momentum, d_z is the distance in z of the helix from the pivot point, and $\tan\lambda$ is the tangent of the dip angle. Note that $\rho = \alpha/\kappa$ is the charge-signed radius of the helix, with $\alpha \equiv 1/(cB)$.

A position on the helix is parameterized in terms of an angle ϕ as follows:

$$\begin{aligned} x(\phi) &= x_0 + d_\rho \cos\phi_0 + \frac{\alpha}{\kappa}\left(\cos\phi_0 - \cos(\phi_0 + \phi)\right) \\ y(\phi) &= y_0 + d_\rho \sin\phi_0 + \frac{\alpha}{\kappa}\left(\sin\phi_0 - \sin(\phi_0 + \phi)\right) \\ z(\phi) &= z_0 + d_z - \frac{\alpha}{\kappa}\tan\lambda \cdot \phi \end{aligned} \tag{1}$$

This is coded into the method "atPhi" of StateVector.java. Similarly, the method "getMom" codes the calculation of the momentum at a point on the helix:

$$\begin{aligned} p_x(\phi) &= -\sin(\phi_0 + \phi)/|\kappa| \\ p_y(\phi) &= -\cos(\phi_0 + \phi)/|\kappa| \\ p_z(\phi) &= \tan\lambda/|\kappa| \end{aligned} \tag{2}$$

These helix parameters differ from those used elsewhere in hps-java $(d_0, \phi_0, \Omega, z_0, \tan\lambda)$, which conform to the LCSim convention. The reason is that I did not realize at first what was in hps-java. The biggest difference is that LCSim uses the inverse of the radius in place

of κ. Another difference is that in LCSim ϕ_0 is the direction of the track propagation at the pivot. I somewhat prefer using κ, as it does not change as the field magnitude changes. At any rate, the transformation between the two is simple once you figure it out. It and its inverse are implemented in the methods "getLCSimParams" and "ungetLCSimParams" in KalmanInterface.java. The transformation also takes into account the Kalman versus HPS-Tracking coordinate system inversion (see below):

$$\begin{aligned} d_0 &\leftarrow d_\rho \\ \phi_0 &\leftarrow -\phi_0 \\ \Omega &\leftarrow -\kappa/\alpha \qquad (3) \\ z_0 &\leftarrow -d_z \\ \tan\lambda &\leftarrow -\tan\lambda \qquad (4) \end{aligned}$$

It can be confusing that the helix parameters are not themselves valid in the global coordinates, which is necessary given the way that they are defined. Instead they are valid in a coordinate system in which the z axis aligns with the local magnetic field, see Section 6.2. The biggest trouble perhaps is with $\tan(\lambda)$, which can be quite different from what one would calculate in global coordinates from $p_z/\sqrt{p_x^2 + p_y^2})$, because except on the magnet symmetry plane the z axis of the local field tends to be tilted. In particular, care with that has to be taken when comparing with GBL fits. Since GBL always assumes a uniform field, with the ideal orientation, then its $\tan(\lambda)$ is always given in global coordinates and will not directly match what comes back from the Kalman code.

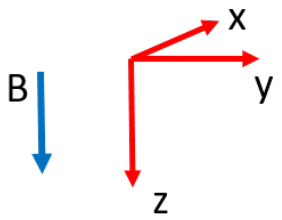
6 Coordinate Systems

There are three coordinate systems used within the code:

- **Global Coordinates:** in these coordinates the z axis is in the nominal magnetic field direction and the y axis is in the nominal beam direction.
- **Local Field Coordinates:** this system is needed because the definition of the helix assumes that the $\vec{B}$ field is aligned with the z axis. Hence in this system the origin is placed at the center of the local silicon-strip detector, the z axis is aligned with the field direction at that location, and the x axis is set perpendicular to the global y axis.
- **Local Sensor Coordinates:** here the z axis is exactly perpendicular to the sensor, and the y axis is perpendicular to the strips, such that the sensor measures this local y coordinate. Instead of (x, y, z), this system is sometimes in the code referred to as (u, v, t).

6.1 Global Coordinates

This system is *not* the HPS global system. It is a system in which $\hat{y}$ is the nominal beam direction and $\hat{z}$ is the nominal $\vec{B}$ field direction.



It is similar to the HPS tracking coordinate system, but unfortunately when I started on the code I didn't realize that in that system the nominal $\vec{B}$ field pointed opposite the z axis. At any rate, the conversion from Kalman global coordinates to HPS global coordinates is as follows:

$$\begin{aligned} x &= x_{\text{hps}} \\ y &= z_{\text{hps}} \\ z &= -y_{\text{hps}} \end{aligned} \tag{5}$$

Furthermore, the conversion from HPS tracking coordinates to Kalman global coordinates is as follows:

$$x = y_{\text{hps-trk}}$$

$$y = x_{\text{hps-trk}} \tag{6}$$

$$z = -z_{\text{hps-trk}} \tag{7}$$

$$\tag{8}$$

These coordinate transformation are implemented in methods of KalmanInterface.java. Also, note that the HPS field map is in HPS global coordinates. The getField method in KalmanInterface.java returns the field in Kalman global coordinates given a position in that same coordinate system.

6.2 Local Field Coordinates

These coordinates exactly align the z axis with the local $\vec{B}$ field, where "local" means the center of the silicon detector that is of immediate interest in the code. The alignment with the field is the whole point of these coordinates; the orientation of the other axes is somewhat arbitrary. I define them such that the local x axis is perpendicular to the global y axis. Mathematically, then, the definition is as follows (with prime designating the local field coordinates):

$$\begin{aligned} \hat{z}' &= \vec{B}/|\vec{B}| \\ \hat{x}' &= (\hat{y} \times \hat{z}')/|\hat{y} \times \hat{z}'| \\ \hat{y}' &= z' \times x' \end{aligned} \tag{9}$$

The origin of this local field system is normally placed at the same origin as the global coordinate system, that is the point $(0, 0, 0)$, which is near the center of the target, but that choice is made by the program that calls Kalman fitting code, e.g. KalmanTrackFit2.java or KalTrack. This transformation is calculated in the constructor of StateVector.java and

stored locally as "Vec origin" and "RotMatrix Rot", where "Vec" is a class of simple vectors and "RotMatrix" is a class of 3-dimensional orthogonal matrices. The methods "toLocal" and "toGlobal" in StateVector.java apply this transformation and its inverse.

The method "rotateHelix" in StateVector.java applies this same transformation to helix parameters, which is a more complicated, nonlinear transformation. It functions by first transforming from the helix parameters to a momentum vector (as in Eqn. 2 with $\phi = 0$), which is easily transformed by the rotation matrix. Then it makes a new helix using the transformed momentum. This requires also transforming the pivot point of the helix, not just the 5 helix parameters. For simplicity, the new pivot point is placed on the helix, where it intersects the silicon. That results in the helix parameters d_ρ and d_z being exactly zero for the transformed helix. The transformation from momentum to helix parameters is

$$\begin{aligned} \phi_0 &= \text{atan2}(-p_x, p_y) \\ \kappa &= Q/\sqrt{p_x^2 + p_y^2} \\ \tan\lambda &= p_z/\sqrt{p_x^2 + p_y^2} \end{aligned} \tag{10}$$

which is coded into the method "pToa" of StateVector.java.

The code also calculates the derivative matrix of this transformation, as it is needed in order to transform properly the covariance matrix of the helix parameters. Since the pivot point is taken to be the origin about which the rotation occurs, then d_ρ and d_z are not affected. Let a and a' be the initial and final subset of helix parameters $(\phi_0, \kappa, \tan\lambda)$, while $\vec{p}$ and $\vec{p}'$ are the momentum and rotated momentum. Then the linearized transformation is a 3×3 matrix according to

$$\frac{\partial a'}{\partial a} = \frac{\partial a'}{\partial p'} \cdot \frac{\partial p'}{\partial p} \cdot \frac{\partial p}{\partial a} \tag{11}$$

Here $\partial p'/\partial p$ is just the rotation matrix described above, which is of course a totally linear transformation. Let Q be the charge sign and $p_t \equiv \sqrt{p_x^2 + p_y^2}$. Then the other derivative matrices are

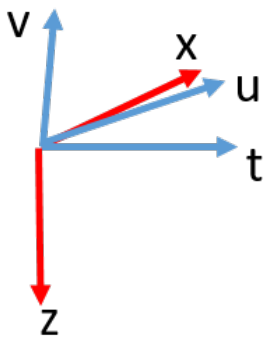
$$\frac{\partial p}{\partial a} = \begin{pmatrix} -\cos\phi_0/|\kappa| & \sin\phi_0/(\kappa \cdot |\kappa|) & 0 \\ -\sin\phi_0/|\kappa| & -\cos\phi_0/(\kappa \cdot |\kappa|) & 0 \\ 0 & -\tan\lambda/(\kappa \cdot |\kappa|) & 1/|\kappa| \end{pmatrix} \tag{12}$$

$$\frac{\partial a'}{\partial p'} = \begin{pmatrix} -p_y/p_t^2 & p_x/p_t^2 & 0 \\ -Qp_x/p_t^3 & -Qp_y/p_t^3 & 0 \\ -p_x p_z/p_t^3 & -p_y p_z/p_t^3 & 1/p_t \end{pmatrix} \tag{13}$$

The full 5×5 transformation is formed by adding in the identity transformations for d_ρ and d_z. I checked this derivative matrix in some special cases by comparing with numerical calculations of small transformation steps. That test code is commented out (and subsequently garbled by Eclipse's annoying propensity to reformat comment lines).

6.3 Local Sensor Coordinates (u, v, t)

The local sensor coordinates are implemented using the class Plane.java, which simply defines a 2D plane in 3D space by means of a location in the plane and the direction cosines of the three axes. The location in the plane is taken to be the center of the silicon wafer. The axes are oriented as indicated in this figure, with $\hat{t}$ being orthogonal to the silicon plane and $\hat{v}$ being orthogonal to the strips on the sensor.



The set of direction cosines defines the orthogonal transformation between the global coordinates and local sensor coordinates. The transformation matrix and its inverse are found in SiModule.java, which also stores the size of the active silicon and a list of all of the hits in that wafer, plus the silicon thickness and a flag indicating whether it is a stereo or axial layer. The methods "toGlobal" and "toLocal" of SiModule.java implement the orthogonal transformations. For example, here is the rotation matrix to go from global to local coordinates in the case of a sensor plane aligned exactly perpendicular to the global y axis in a stereo layer with stereo angle θ:

$$R = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ -\sin\theta & 0 & -\cos\theta \\ 0 & 1 & 0 \end{pmatrix} \tag{14}$$

In reality the sensors are not so aligned, because the SVT is rotated about the vertical axis by 30.5 mrad and because there are numerous small alignment corrections. The rotation matrix and translation vector for each sensor are copied directly from hps-java and transformed to Kalman coordinates by methods of KalmanInterface.java.

Unfortunately, the local sensor coordinates in hps-java are defined a bit differently. The correspondence is as follows:

$$\begin{aligned} u_{\text{hps}} &= v_{\text{kal}} \\ v_{\text{hps}} &= -u_{\text{kal}} \\ w_{\text{hps}} &= t_{\text{kal}} \end{aligned} \tag{15}$$

(16)

7 HelixState Class

Helices are specified using the class HelixState. It includes the helix parameters, as defined in Section 5, as well as their covariance matrix and the pivot point to which the helix parameters apply. Furthermore, it also includes the definition of the field-aligned coordinate system within which the helix parameters are valid. To construct a HelixState instance one needs to supply the following:

- the five helix parameters
- the covariance matrix of the helix parameters

- the pivot point
- an origin of the coordinate system in which the helix parameters and pivot are valid (see below)
- the local magnetic field vector

As always, the helix parameters and pivot point are assumed to be valid only in a coordinate system with z axis aligned with the local magnetic-field vector and with its origin (in global coordinates) located as specified. Note that the origin and pivot point can be the same physical point, in which case the pivot has coordinates $(0, 0, 0)$.

8 Track Propagation

8.1 Analytic Helix-Plane Intersection

The method "planeIntersect" in HelixPlaneIntersect.java does an analytic calculation of the intersection of a helix with a more-or-less arbitrary plane. In truth, it is optimized for planes close to being perpendicular to the beam axis (y), as is the case the HPS silicon detector planes. If a detector plane lies exactly in a plane of constant y, in the local coordinate system of the helix, then calculating the intersection is very simple and fast. One simply inverts the $y(\phi)$ equation in Eqn. 1 and solves for ϕ given the y of the detector plane. The code uses this as a starting guess and then numerically solves the more general problem. If $\vec{r}(\phi)$ is the point given by Eqn. 1, $\vec{r}_0$ is a point in the plane (e.g. the center of the silicon), and $\hat{t}$ is a unit vector perpendicular to the plane, then

$$f(\phi) \equiv (\vec{r}(\phi) - \vec{r}_0) \cdot \hat{t} \tag{17}$$

will be zero when ϕ is the solution for the intersection. A Newton-Raphson zero finding algorithm from Numerical Recipes in C is used to find the zero of $f(\phi)$ in the neighborhood of the initial guess (method "rtSafe" in HelixPlaneIntersect.java). Usually just one or two iterations is enough, because the HPS planes are quite close to being in planes of constant y. Therefore the calculation is reasonably efficient.

8.2 Runge-Kutta Integration

The analytic helix propagation is not suitable for extrapolating particles back to the target or to the calorimeter. It is also not useful for simulating particle trajectories in the fast stand-along simulation. For these purposes the code has methods to solve the interaction between fast charged particle and magnetic field by fourth-order Runge-Kutta integration of the differential equation (i.e. Newton's second law, using the Lorentz force law). The class RungaKutta4.java does the integration through the HPS field map. Its constructor must be provided with the charge sign, the step size, and the field map. The method "integrate" is then called with the initial position and momentum (in the global coordinates of the field map) and a total distance to propagate. It returns a six-element vector. The first three components are the final position, and the last three are the final momentum. The method is quite simple and works only with a constant step size.

The method "rkIntersect" in HelixPlaneIntersect.java makes use of the Runge-Kutta integration to find the intersection of a particle trajectory with a silicon plane. It is designed

to work well only for the low curvature tracks of interest in HPS. It first estimates the distance to the plane by taking a straight line path from the initial point to the plane. Then it calls the Runge-Kutta integrate method to propagate that distance. At the end of the integration it calculates new helix parameters and then calls the analytic routine to find the intersection of the helix with the plane. The idea is that the Runge-Kutta integration should get close enough to the plane that the error from the helix approximation of the remainder of the trajectory will be negligible. This saves testing at every integration step whether the plane has been crossed.

In the class HelixState.java, the method "propagateRungeKutta" is used to propagate the result of the Kalman-Filter fit to a plane containing the origin (i.e. the target), or to some other plane such as the face of the electromagnetic calorimeter. It is used in KalTrack to propagate the helix from the first silicon-strip plane back to the origin, through a fairly non-uniform magnetic field. The "rkIntersect" method described above readily does most of that job. The problem then is to extrapolate the covariance matrix. Trying to do that rigorously for each numerical-integration step seemed to be overkill, whereas the covariance propagation by means of a pivot transformation is already built into the Kalman-Filter code (see Section 9). Approximating the covariance propagation by a single pivot transformation from the first layer to the origin turned out not to be quite good enough, so a new method named "helixStepper" in HelixState.java was written to do the transformation in multiple small helix steps, reorienting with the local field after each step. This method also accounts for multiple scattering in each intervening layer of silicon, as long as the user passes a list of y locations of the relevant silicon planes.

To use the propagateRungeKutta method outside of KalTrack, one has to construct a HelixState instance and then pass the plane to which to extrapolate, an array of y values where silicon is located (which can be approximate, as it is only used to propagate the covariance), the thickness of the silicon in radiation lengths, and the field map. The method propagateRungeKutta then returns a new HelixState instance, for which the origin will be located at the point specified as the origin of the plane supplied in the call to propagateRungeKutta. The pivot will be the point at which the particle's trajectory intersects the plane. HelixState has methods then to return points on the helix, as a function of the turning angle ϕ, but it has to be understood that those points will be in the local magnetic-field coordinate system. There also are methods to convert them to global coordinates.

9 Pivot Transformation

A single helix is described by a pivot point and five helix parameters (see Section 5). Since the pivot point is an arbitrary choice, and each choice results in a unique set of helix parameters, there must be a relation between helix parameters for one pivot versus another. That relation is called a "pivot transformation." It is implemented by the method "pivotTransform" in StateVector.java. The method is overloaded with several calling sequences, all of which eventually call a static method that contains the essential equations. If $\vec{r}_0 = \{x_0, y_0, z_0\}$ is the starting pivot and $a = \{d_\rho, \phi_0, \kappa, d_z, \tan\lambda\}$ is the set of helix parameters for that pivot,

then the code starts by finding the center of the helix according to

$$\begin{aligned} x_c =& x_0 + \left(d_\rho + \frac{\alpha}{\kappa}\right)\cos\phi_0 \\ y_c =& y_0 + \left(d_\rho + \frac{\alpha}{\kappa}\right)\sin\phi_0 \end{aligned} \tag{18}$$

With $\vec{r}' = \{x_0', y_0', z_0'\}$ being the new pivot, the new helix parameters, then, are calculated according to

$$\begin{aligned} d_\rho' =& (x_c - x_0')\cos\phi_0' + (y_c - y_0')\sin\phi_0' - \frac{\alpha}{\kappa} \\ \phi_0' =& \text{atan2}(y_c - y_0', x_c - x_0') \quad (\kappa > 0) \\ =& \text{atan2}(y_0' - y_c, x_0' - x_c) \quad (\kappa < 0) \\ \kappa' =& \kappa \\ d_z' =& z_0 - z_0' + d_z - \frac{\alpha}{\kappa}(\phi_0' - \phi_0)\tan\lambda \\ \tan\lambda' =& \tan\lambda \end{aligned} \tag{19}$$

In order to implement the Kalman filter, including propagation of the covariance matrix, the derivative matrix of the transformation 19 is also needed. It is coded into the static method "makeF" of StateVector.java, which returns a matrix of the class SquareMatrix.java. The nonzero elements of the derivative matrix $\mathbf{F}$ are

$$\begin{aligned} f_{00} =& \cos(\phi_0' - \phi_0) \\ f_{01} =& \left(d_\rho + \frac{\alpha}{\kappa}\right)\sin(\phi_0' - \phi_0) \\ f_{02} =& \left(\frac{\alpha}{\kappa^2}\right)\cdot(1 - \cos(\phi_0' - \phi_0)) \\ f_{10} =& -\frac{\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa} \\ f_{11} =& \frac{(d_\rho + \alpha/\kappa)\cos(\phi_0' - \phi_0)}{(d_\rho' + \alpha/\kappa)} \\ f_{12} =& \left(\frac{\alpha}{\kappa^2}\right)\frac{\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa} \\ f_{22} =& 1 \\ f_{30} =& \left(\frac{\alpha}{\kappa}\right)\frac{\tan\lambda\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa} \\ f_{31} =& \left(\frac{\alpha}{\kappa}\right)\tan\lambda\left(1 - \frac{(d_\rho + \alpha/\kappa)\cos(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa}\right) \\ f_{32} =& \left(\frac{\alpha}{\kappa^2}\right)\tan\lambda\left(\phi_0' - \phi_0 - \frac{(\alpha/\kappa)\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa}\right) \\ f_{33} =& 1 \\ f_{34} =& -\left(\frac{\alpha}{\kappa}\right)\cdot(\phi_0' - \phi_0) \\ f_{44} =& 1 \end{aligned} \tag{20}$$

The pivot transform is used in the Kalman-filter prediction step, to transform the helix parameters from one silicon layer to the next. To account for field non-uniformity, at each layer the pivot transform is followed by a rotation into a coordinate system aligned with the field at that plane. The "rotation" of the helix parameters is described in some detail in Section 6.2. It is implemented in method "rotateHelix" in StateVector.java, which returns not just the transformed helix parameters but also the derivative matrix of that transformation, which then multiplies **F** from Eqn. 20 to produce the complete transformation matrix.

10 Multiple-Scattering Matrix

When propagating the covariance matrix from one silicon plane to the next, multiple scattering is taken into account, in addition to the pivot transform and coordinate rotation. The effect of multiple scattering is simple, as it occurs discretely in the silicon planes, with no material present between planes. Therefore, it simply changes the two angles, ϕ_0 and λ, that specify the helix direction at pivot point, which is always located in the scattering plane. The nonzero elements of the matrix **Q** are

$$\begin{aligned} q_{11} &= \left(1 + \tan^2\lambda\right) \cdot \sigma^2 \\ q_{44} &= \left(1 + \tan^2\lambda\right)^2 \cdot \sigma^2 \end{aligned} \tag{21}$$

where σ is calculated from the usual formula for Gaussian multiple scattering in a thickness d of material with radiation length X_0:

$$\sigma = \frac{0.0136}{|p|}\sqrt{\frac{d}{X_0}} \cdot (1 + 0.038\ln(d/X_0)) \tag{22}$$

The thickness d is calculated from the silicon thickness divided by the cosine of the angle between the track and the silicon, and the radiation length is taken to be about 93.7 mm.

11 The Kalman Filter

The Kalman Filter fitting code follows the formalism of R. Früwirth in "Applications of Kalman Filtering to Track and Virtex Fitting," Nucl. Instr. Meth. A262 (1987) p. 444. It consists of three steps: "prediction," "filtering," and "smoothing." Because the prediction step is nonlinear (the helix is not a linear function), this is what Früwirth calls an "extended" Kalman filter. Due to that non-linearity, the result is not independent of the initial guess for the helix parameters (see Section 12) that is needed in order to get the filter started. Therefore, some improvement can be had by iterating the filter, using the result of the first iteration as the initial guess for the second. I have typically been running it for two iterations. A third tends to give little or no improvement.

11.1 Interface to the Kalman Filter Fit

There are two points of interface into the Kalman Filter fitting:

1. Set up the data that are to be fit by creating multiple instances of the class SiModule.java. Each SiModule contains the relevant data from a single silicon-strip detector.

One has to provide the layer number and number of the detector in that layer, a plane (instance of Plana.java) that represents the center of the detector and its orientation, the detector width and height, the silicon thickness, the magnetic field map, and a flag telling whether it is considered to be an axial or stereo layer. Then one must make calls to the method "addMeasurement" to fill in data for the hits in that detector. Only a single hit should be entered in each SiModule if only track fitting is to be done (i.e. no pattern recognition).

2. Call KalmanTrackFit2.java to execute the track fit or else KalmanPatRecHPS.java to execute the full-blown pattern recognition.

11.2 Prediction

The prediction step is done by the "makePrediction" method of MeasurementSite.java. It in turn calls the "predict" method of StateVector.java. Essential elements of the pattern recognition process are also implemented in the makePrediction method. For example, that is where a search is made for silicon-strip hits that match up with the extrapolated track, followed by selection of the best hit, to be added to the track candidate.

11.2.1 MeasurementSite:makePrediction

The "makePrediction" method typically is handed the filtered state vector at the previous measurement site, together with the SiModule of that site. It then calculates the predicted state vector of "this" site as follows:

- Extrapolate the helix of the input filtered state vector to find where it intersects the plane of the SiModule in this MeasurementSite. To do so, it uses the methods presented in Section 8.

- Only in the case that pattern recognitions is being done, it checks whether the extrapolation is within the bounds of the silicon detector.

- Call the "predict" method of the input StateVector, handing it the intersection point to use as a new pivot. To do this it first has to calculate the thickness of the silicon in which the particle will scatter during the propagation, which is calculated by the silicon thickness divided by the cosine of the angle between the momentum and the normal direction. It also has to get the magnetic field from the map, to pass to the "predict" method, which returns the new predicted state vector at "this" MeasurementSite. The details of what this method does internally are listed below, at the end of this section.

- Call the method "h" to calculate the predicted measurement. This is just the intersection point already calculated transformed into the detector coordinate system. From this, the residual $r = m - m_p$ of the prediction m_p with respect to the measured point m is calculated. In the case that pattern recognitions is being done, at this point the code loops over the (unused) hits in the SiModule and finds the one with the minimum residual.

- For the Kalman Filter to be able to calculate the uncertainties on the predicted measurements, the derivative matrix of "h" also has to be calculated. That is done by

the method "buildH" and is a bit complicated but described in detail below. Since there is only a single measured coordinate in each plane, the derivative "matrix" **H** is really just a 5-component vector. Because of the coordinate transformations used to deal with the non-uniform magnetic field, which always place the pivot point on the helix exactly where it intersects with the detector plane (that is, right at the location of the predicted measurement), then the **H** for the prediction step ends up being fairly trivial, with non-zero elements only for d_ρ and d_z. For the filter and smoothing steps the other elements become non-zero but still quite small.

- With the derivatives in hand, then the covariance of the predicted residual and contribution to the χ^2 may be calculated. First, the covariance:

$$\sigma_r^2 = \sigma^2 - \sum_{ij} H_i C_{ij} H_j \tag{23}$$

where σ is the measurement uncertainty from the silicon-strip detector itself. The minus sign follows from the fact that the fitted track should tend to be pulled, on average, closer to the measured points, compared to the true particle trajectory. That leaves open the possibility of getting nonsensical negative values if the input is not reasonable. In fact, that occurs frequently, but only at the beginning of the fit. It happens then because the "guess" for the initial covariance matrix is made to be very large, so that it does not affect significantly the final result for the covariance and χ^2. Finally, the prediction χ^2 contribution of the given layer is

$$\chi_+^2 = \frac{r^2}{\sigma_r^2} \tag{24}$$

11.2.2 The Derivatives H_i

The derivatives in **H** are calculated as follows. The method "buildH" returns the 5-vector

$$H_i = \frac{\partial m}{\partial a_i} \tag{25}$$

where m is the predicted measurement and **a** is the vector of helix parameters. To calculate this, first let $\mathbf{R}_t$ be the rotation from the field coordinates in which the helix is defined to the detector coordinates. It is calculated from the matrix product

$$\mathbf{M} = \mathbf{R}\mathbf{R}_f^{-1} \tag{26}$$

where **R** is the rotation from global coordinates to detector coordinates, stored in the SiModule instance, and $\mathbf{R}_f$ is the rotation from global coordinates to field coordinates, as stored in the StateVector instance. Since in the detector coordinate system the y coordinate is the measurement, we have

$$\frac{\partial m}{\partial a_j} = \sum_i M_{1j} \frac{\partial x_i}{\partial a_j} \tag{27}$$

where $\vec{x}$ is the location of the intersection point in field coordinates.

If we know the ϕ of the intersection point, then we get the point $\vec{x}$ from Eqn 1, which we can represent as a vector function $\vec{x}(\phi(\mathbf{a}), \mathbf{a})$. In this notation, then, we have

$$\frac{\partial x_i(\phi(\mathbf{a}), \mathbf{a})}{\partial a_j} = \frac{\partial x_i}{\partial \phi} \frac{\partial \phi}{\partial a_j} + \frac{\partial x_i}{\partial a_j} \tag{28}$$

As discussed in Section 8.1, the point $\vec{x}$ is calculated from setting Eqn. 17 equal to zero. We can represent this as $f(\vec{x}(\phi(\mathbf{a}), \mathbf{a})) = 0$, which can be differentiated implicitly to find $\partial\phi/\partial a_j$:

$$\frac{\partial f}{\partial a_j} = \sum_i \frac{\partial f}{\partial x_i} \frac{\partial x_i(\phi(\mathbf{a}), \mathbf{a})}{\partial a_j} = \sum_i \hat{t}_i \left(\frac{\partial x_i}{\partial \phi} \frac{\partial \phi}{\partial a_j} + \frac{\partial x_i}{\partial a_j} \right) = 0 \tag{29}$$

From this it follows that

$$\frac{\partial \phi}{\partial a_j} = -\frac{\sum_i \hat{t}_i \frac{\partial x_i}{\partial a_j}}{\sum_i \hat{t}_i \frac{\partial x_i}{\partial \phi}} \tag{30}$$

where as a reminder, the $\hat{t}_i$ are the direction cosines of the detector plane in the B-field coordinates. So, to calculate $\mathbf{H}$, Eqn. 30 is substituted into Eqn. 28, which in turn is substituted into Eqn. 27.

The needed derivatives are

$$\begin{aligned} \frac{\partial x_0}{\partial \phi} &= \frac{\alpha}{\kappa} \sin(\phi_0 + \phi) \\ \frac{\partial x_1}{\partial \phi} &= -\frac{\alpha}{\kappa} \cos(\phi_0 + \phi) \\ \frac{\partial x_2}{\partial \phi} &= -\frac{\alpha}{\kappa} \tan\lambda \end{aligned} \tag{31}$$

and a 3×5 matrix for $\partial x_i/\partial a_j$, for which the nonzero elements are

$$\begin{aligned} \frac{\partial x_0}{\partial a_0} &= \cos\phi_0 \\ \frac{\partial x_1}{\partial a_0} &= \sin\phi_0 \\ \frac{\partial x_0}{\partial a_1} &= -\left(d_\rho + \frac{\alpha}{\kappa}\right) \sin\phi_0 + \frac{\alpha}{\kappa} \sin(\phi_0 + \phi) \\ \frac{\partial x_1}{\partial a_1} &= \left(d_\rho + \frac{\alpha}{\kappa}\right) \cos\phi_0 - \frac{\alpha}{\kappa} \cos(\phi_0 + \phi) \\ \frac{\partial x_0}{\partial a_2} &= -\frac{\alpha}{\kappa^2} (\cos\phi_0 - \cos(\phi_0 + \phi)) \\ \frac{\partial x_1}{\partial a_2} &= -\frac{\alpha}{\kappa^2} (\sin\phi_0 - \sin(\phi_0 + \phi)) \\ \frac{\partial x_2}{\partial a_2} &= \frac{\alpha}{\kappa^2} \tan\lambda \cdot \phi \\ \frac{\partial x_2}{\partial a_3} &= 1 \\ \frac{\partial x_2}{\partial a_4} &= -\frac{\alpha}{\kappa} \cdot \phi \end{aligned} \tag{32}$$

11.2.3 StateVector:predict

What goes on in the "predict" method of StateVector.java is as follows:

- A new state vector is made with helix parameters that are transformed to correspond to the new pivot point, using Eqn. 19. The derivative matrix $\mathbf{F_p}$ of the pivot transform is also calculated.

- The pivot point is then transformed to the global reference system using the "toGlobal" method of the old StateVector instance, and then it is transformed to the field system at the new silicon plane using the "toLocal" method of the new StateVector instance. The new coordinate system has its origin at the center of the silicon detector and is oriented such that the field at that point is parallel to the $\hat{z}$ axis.

- The helix parameters are then rotated to the new coordinate system, again using a composition of the rotations from the old system to the global system and then to the new local system. That rotation is done by the "rotateHelix" method, as described in Section 6.2. Since the pivot point is always placed exactly on the predicted helix, the d_ρ and d_z parameters are zero both before and after the coordinate rotation. They do not change at all, which simplifies the transformation.

- The "rotateHelix" method also returns a derivative matrix of that helix-rotation transformation, which then multiplies $\mathbf{F_p}$ to give the overall derivative matrix $\mathbf{F}$ for the composition of the pivot transformation and coordinate transformation.

- The covariance matrix $\mathbf{C}$ of the helix parameters is increased by adding in the multiple-scattering contribution $\mathbf{Q}$ from Eqn. 21, which increases the errors on ϕ_0 and $\tan\lambda$.

- Then the covariance matrix is transformed to the new pivot point and the new coordinate system by $\mathbf{F}$:

$$\mathbf{C}' = \mathbf{F}^T\mathbf{C}\mathbf{F} \tag{33}$$

11.3 Filtering

After each prediction step, a filter step is executed to update the helix parameters according to the measurement found in the new silicon-strip layer. The "filter" method of MeasurementSite.java first calls the "filter" method of the predicted StateVector instance and then calculates the filtered residual.

The StateVector.java "filter" method does most of the work. It uses the "gain matrix" formalism of Früwurth, as follows (although I checked that the weighted-means formalism gives identical results). With $\mathbf{H}$ being the 5-vector calculated in the prediction stage and passed to this "filter" method from the MeasurementSite method, the "gain matrix" $\mathbf{K}$, which in this case is just a 5-vector, is calculated as

$$K_i = \frac{\sum_j C_{ij}H_j}{\sigma^2 + \sum_{ij} H_i C_{ij} H_j} \tag{34}$$

where σ is the point measurement uncertainty of the silicon-strip detector.

The filtered helix parameters then are

$$a_{f_i} = a_i + K_i \cdot r \tag{35}$$

where r is the predicted residual. Similarly the filtered covariance matrix of the helix parameters is calculated:

$$C_{f_{ij}} = \sum_m \left(I_{im} + K_i H_m\right) C_{mj} \tag{36}$$

where I is the identity matrix.

MeasurementSite.filter then calculates the predicted measurement and residual using the h function, just as in the prediction step. The derivatives H_i are recalculated using the filtered helix parameters (which makes only fairly small changes relative to the previously calculated H_i). Finally, the covariance of the filtered residual and the χ^2 contribution are calculated just as in Eqns 23 and 24.

11.4 Smoothing

The smoothing step is done after all the prediction/filter steps are done and the program has arrived at the last layer. It then steps in reverse, going layer by layer back to the track beginning, executing the smoothing step at each layer. The smoothed helix parameters at each layer represent the best estimate of the particle's trajectory at that layer, based on information from hits in all of the layers.

The "smooth" method of MeasurementSite.java first calls the method of the same name of the filtered instance of StateVector.java, which does most of the work. Let $\mathbf{C_p}'$ be the covariance matrix of the predicted state vector of the previous site that was smoothed, while $\mathbf{C_f}$ is the covariance matrix of the filtered state that is to be smoothed, and $\mathbf{F}$ is the propagator matrix. First, we calculate a matrix $\mathbf{A}$ from

$$A_{ij} = \sum_{mn} C_{f\,im} F_{nm} {C'_p}^{-1}{}_{nj} \tag{37}$$

Then the smoothed helix parameters are

$$a_{si} = a_{f\,i} + \sum_j A_{ij} \left(a'_{sj} - a'_{p\,j} \right) \tag{38}$$

where a'_{sj} and $a'_{p\,j}$ are respectively the smoothed and predicted helix parameters of the previously smoothed state. Similarly, the covariance of the smoothed helix parameters is

$$C_{sij} = C_{f\,ij} + \sum_{mn} A_{mi} \left(C'_{smn} - C'_{p\,mn} \right) A_{nj} \tag{39}$$

where $\mathbf{C}'_s$ is the covariance of the smoothed helix parameters of the previously smoothed site.

With that done, then MeasurementSite.smooth uses the "h" and "H" methods with the smoothed helix in order to calculated the smoothed residual, its covariance, and the χ^2 contribution in exactly the same way that it was done for the predicted and filtered states. It may be a waste of CPU time to recalculate "H", since the changes are generally quite small. However, I did find that it makes a visible difference in the last couple of layers, where the field non-uniformities are relatively large. Without recalculating the derivatives using the smoothed helix parameters, there is a rather large tail on the residuals distributions that mostly goes away if the recalculation is done.

12 Seeding the Kalman Filter

The Kalman Filter requires an initial guess for the helix and covariance matrix before it can begin fitting a track. The initial guess cannot be totally random in this case, in which

the fit is inherently non-linear, or else many iterations may be required to converge to a good result. On the other hand, the covariance matrix should be very large, such that the covariance of the initial guess, if it is based on the tracking hits themselves, does not influence the final covariance matrix. Otherwise the hits used to generate the initial guess will get double counted in the error estimations. Unfortunately, blowing the covariance matrix up too large frequently results in numerical problems and non-positive-definite covariance matrices from the fit. The factors by which the seed covariance matrix is inflated before starting the Kalman fit were chosen with those competing issues in mind.

What the present code does, in class SeedTrack.java, is make a linear fit to five or more strip hits (at least two axial and three stereo) to generate the initial guess. That fit generates a real covariance matrix, but the elements of the matrix are inflated by a factor before feeding it to the Kalman Filter routine, as mentioned above. Normally the Kalman filter is iterated in a second pass, using the result of the first pass to initialize the second, again inflating the covariance matrix before starting the Kalman Filter. Additional iterations beyond that do not seem to add significant value.

For simplicity, SeedTrack.java assumes that the magnetic field is exactly aligned with the global z axis. For the field magnitude it takes the average value from the field map, averaged over the planes containing the hits to be used for fitting the seed.

The track model used for the seed fit is a straight-line in the non-bending plane (y, z) and a parabola in the bending plane (x, z). The two have to be fit simultaneously, however, as the silicon strips in general do not measure x or y. They could be fit separately if 3-D hits were first calculated, but that has a number of disadvantages that I wanted to avoid. The 5-parameter linear fit is quite fast, anyway. The job is done by the LinearHelixFit method in SeedTrack.java. For each of the N hits (typically 3 stereo and 2 axial), the method must be provided with

- $\vec{\delta}$, the offset of the local detector coordinate system from the origin of the global coordinate system.
- $\mathbf{R}$, the rotation matrix from the global system to the local detector system (which includes the stereo angle). Actually, only the second row of the matrix is needed, in order to calculate the predicted v coordinate in the detector system.
- $v_{\rm m}$, the measured value of the coordinate in the detector system, from the strips that were hit.
- s, the one-sigma error estimate on v.
- y, the nominal y coordinate of the hit in the global system (i.e. along the beam axis), calculated by taking the hit to be at the center of the strip in the detector system and transforming that point back to the global system.

With the line parameterized as $z = a+by$ and the parabola as $x = c+dy+ey^2$, the predicted hit location at a given layer is

$$v_{\rm pred} = R_{10}\left[c + dy + ey^2 - \delta_0\right] + R_{11}\left[y - \delta_1\right] + R_{12}\left[a + by - \delta_2\right]. \tag{40}$$

The fit then minimizes the χ^2:

$$\chi^2 = \sum_{j=1}^{N} \left[\frac{v_{{\rm m}i} - v_{{\rm pred}_i}}{s_i}\right]^2, \tag{41}$$

which leads to the linear equation to be solved $\mathbf{A}\vec{x} = \vec{b}$, with $\vec{x} \equiv \{a, b, c, d, e\}$ and

$$\mathbf{A} = \sum_{i=1}^{N} \begin{pmatrix} w_i R_{12i}^2 & w_i y_i R_{12i}^2 & w_i R_{12i} R_{10i} & w_i y_i R_{12i} R_{10i} & w_i y_i^2 R_{12i} R_{10i} \\ w_i y_i R_{12i}^2 & w_i y_i^2 R_{12i} R_{10i} & w_i y_i R_{12i} R_{10i} & w_i y_i^2 R_{12i} R_{10i} & w_i y_i^3 R_{12i} R_{10i} \\ w_i R_{12i} R_{10i} & w_i y_i R_{12i} R_{10i} & w_i R_{10i}^2 & w_i y_i R_{10i}^2 & w_i y_i^2 R_{10i}^2 \\ w_i y_i R_{12i} R_{10i} & w_i y_i^2 R_{12i} R_{10i} & w_i y_i R_{10i}^2 & w_i y_i^2 R_{10i}^2 & w_i y_i^3 R_{10i}^2 \\ w_i y_i^2 R_{12i} R_{10i} & w_i y_i^3 R_{12i} R_{10i} & w_i y_i^2 R_{10i}^2 & w_i y_i^3 R_{10i}^2 & w_i y_i^4 R_{10i}^2 \end{pmatrix} \tag{42}$$

where $w_i \equiv 1/s_i^2$, and with

$$\vec{b} = \sum_{i=1}^{N} \begin{pmatrix} v_{ci} R_{12i} w_i \\ v_{ci} y_i R_{12i} w_i \\ v_{ci} R_{10i} w_i \\ v_{ci} y_i R_{10i} w_i \\ v_{ci} y_i^2 R_{10i} w_i \end{pmatrix} \tag{43}$$

where $v_{ci} \equiv v_i + R_{10}\delta_{0i} + R_{11}(\delta_{1i} - y_i) + R_{12}\delta_{2i}$.

The solution vector $\vec{x}$ and its covariance C must then be converted to helix parameters and their covariance in SeedTrack.java. The method "parabolaToCircle" converts the three parabola coefficients $\{c, d, e\}$ to the helix parameters $\{d_\rho, \phi_0, \kappa\}$. First, the radius of the circle is $R = 1/(2e)$. The center of the circle is

$$\begin{aligned} x_c &= R \cdot d \\ y_c &= c + R\left(1 - d^2/2\right) \end{aligned} \tag{44}$$

The helix parameters, then, are

$$\begin{aligned} \phi_0 &= \text{atan2}\left(y_c, x_c\right) \\ \kappa &= \frac{\alpha}{R} \\ d_\rho &= \frac{x_c}{\cos\phi_0} - R \\ \tan\lambda &= a \cdot \cos\phi_0 \\ d_z &= b + d_\rho \tan\lambda \tan\phi_0 \end{aligned} \tag{45}$$

Some derivatives for transforming the convariance are

$$\begin{aligned} \frac{d\phi_0}{dc} &= \frac{2e}{d}\sin^2\phi_0 \\ \frac{d\phi_0}{dd} &= \frac{\sin\phi_0\cos\phi_0}{d - \sin^2\phi_0} \\ \frac{d\phi_0}{de} &= \frac{2c}{d}\sin^2\phi_0 \end{aligned} \tag{46}$$

If C_f is the covariance of the linear fit, then the covariance of the helix parameters is

$$C = D^T C_f D \tag{47}$$

where the non-zero values of the derivative matrix D are

$$
\begin{aligned}
D_{02} &= \frac{1}{\cos\phi_0} + x_c \frac{\tan\phi_0}{\cos\phi_0} \cdot \frac{d\phi_0}{dc} \\
D_{03} &= -\left(\frac{d}{2e\cos\phi_0}\right) + x_c \frac{\tan\phi_0}{\cos\phi_0} \cdot \frac{d\phi_0}{dd} \\
D_{04} &= \left[1 - \left(\frac{1 - d^2/2}{\cos\phi_0}\right)\right] / (2e^2) + x_c \frac{\tan\phi_0}{\cos\phi_0} \cdot \frac{d\phi_0}{de} \\
D_{12} &= \frac{d\phi_0}{dc} \\
D_{13} &= \frac{d\phi_0}{dd} \\
D_{14} &= \frac{d\phi_0}{de} \\
D_{24} &= 2\alpha \\
D_{30} &= 1 \\
D_{31} &= d_\rho \sin\phi_0 \\
D_{41} &= \cos\phi_0
\end{aligned}
\tag{48}
$$

(49)

Finally, the results have to be rotated into the frame of the local magnetic field. This primarily affects $\tan\lambda$, which follows directly the tilt of the magnetic field.

13 Kalman Filter Refit of an Existing Track

The class KalmanTrackFit2 is used to do a refit of a given HPS track without doing any pattern recognition (i.e. not changing any hit assignments), a process that is mainly useful for debugging and evaluation and is unlikely to be part of the HPS reconstruction flow. Before calling KalmanTrackFit2, an array of SiModule instances must be created, one for a single silicon-strip detector in each layer. Then, in each SiModule the hit list must be filled with a single hit, that is, an instance of Measurement.java, as discussed above in Section 11.1. The list is passed to KalmanTrackFit2.java, along with an initial helix guess: pivot point, helix parameters, and covariance matrix. Normally the guess will come from the code described in Section 12, except that the covariance matrix elements should all be scaled up by at least a factor of 1000 (otherwise the information from the hits used to generate the initial guess gets double counted in the statistical errors). From the initial guess the code creates a StateVector instance to use for getting the Kalman filter started. KalmanTrackFit2 also requires input of the starting point in the list of SiModules and the number of iterations.

When pattern recognition is not being done, then it makes sense just to start at the first layer. In that case the code will step through the layers, doing a prediction to the next layer and then filtering that layer, until it gets to the other end of the detector. Then it does the smoothing layer by layer until it gets back to the start. The result, taken to be the smoothed state at the first layer [1] and is stored as an instance of the KalTrack class.

[1] Of course, if one wants to extrapolate to the ECal, then the smoothed or filtered state at the last layer would be used as the starting point.

The method "originHelix" of KalTrack.java is called to extrapolate the track to the vertex region. It uses the "propagateRungeKutta" method of StateVector.java to do that job, as described in Section 8. There also are methods in KalTrack.java to calculate "kinks" between layers. Those are found by extrapolating the smoothed helix parameters from two adjacent layers to a common plane and then calculating the relative angles in two projections.

14 Pattern Recognition using the Kalman Filter

The pattern recognition code is for the most part contained in the class KalmanPatRecHPS, called from the method KalmanPatRec in KalmanInterface.java. Its output is an array-list of KalTrack objects, one for each found track. Those output tracks are meant to be independent and unique. They may share a few hits, but only in cases in which the hit is so close to both tracks that it might be a true overlapping hit. The KalTrack objects get transformed into HPS tracks by the createTrack method in KalmanInterface.java.

The input is an array-list of SiModule objects, one for each silicon-strip detector with hits, and the event number, to identify the event. The code initializes by making array-lists of hits (KalHit objects) in each layer and modules in each layer. It then works from an array-list of "seed strategies." Each strategy is a list of 5 layers, two axial and three stereo, to be used for forming track seeds, each of which can then be used to initiate the Kalman filter. The seed strategies and a rather large list of other parameters to control the pattern recognition are contained within the class KalmanParams.java. Default values are hard coded there, but most of them can be changed from the steering file for running the driver KalmanPatRecDriver.java.

The pattern recognition is then contained within two large nested loops. First there is a global loop over iterations (so far limited to two). The idea is to use tight tracking cuts in the first iteration and looser cuts in the second, in order to give the best tracks higher priority in taking hits. I thought that the number of global iterations could probably be reduced to one to save CPU time, without losing much, but in fact I found that it tended to increase the CPU time instead, or at best hold it about constant. Second there is a loop over the seed strategies, each of which is a set of three stereo layers and two axial layers.

For each seed strategy, then, the code begins the pattern recognition by looping over all combinations of hits in the five seed layers, skipping hits on tracks finalized in the previous iteration. Optionally, hits with amplitude below a set threshold are not used on the seeds, which can greatly decrease the number of combinations and resulting track candidates to fit when there is a lot of noise in the event. Similarly, hit combinations are skipped if they encompass a time range greater than a certain cut value, and combinations are also skipped if all 5 hits are already contained within an existing track candidate. To save some time, seeds are also skipped if pairs of hits on axial layers extrapolate too far away from the beam vertex.

For seeds not already rejected, the code calls SeedTrack (see Section 12) for each combination, to do a linear 5-parameter fit to the 5 hits. Since there are no degrees of freedom in the fit, there can be no cut on the fit quality. However, the code does cut on the track curvature κ, the $\tan(\lambda)$, and the distance of the track from the origin in the plane of the target. At the same time, cuts out near duplicates, defined as seeds that have nearly the same fit parameters as seeds already in the list of accepted seeds.

The accepted seeds then get sorted for quality (based on their projected distances from

the origin), and starting with the best seed, the Kalman filter is used to try to build a full track. (Subsequent seeds get skipped if they are entirely contained within an already formed candidate track.) First, the method "filterTrack" is called to fit the five seed hits plus hits in all layers from the seed beginning out to the farthest layer downstream. It keeps the hit assignments in the seed layers, while in the other layers it takes the hit closest to the track extrapolation, except that it skips hits used by finalized tracks from a previous global iteration, if there was one. Optionally, it can give preference to high-amplitude hits, to avoid picking up low-amplitude noise hits. Nevertheless, low-amplitude hits will get picked up if nothing better is available.

If the results of the outward filtering are acceptable, then the code does the Kalman smoothing operation, using the method "smoothTrack," back to the beginning of the seed. From that point it starts doing the Kalman filter inward, toward the target, to account for the rest of the SVT layers. It compares the result against previous candidates and skips to the next seed if the result is completely redundant with a set of hits already tried (*i.e.* the hit selections are identical). Otherwise the candidate gets added to the list. Some quality cuts are made to select "good" candidates at this point. If the fit chi-squared is too large, the code tries to see if it can be significantly improved by removing a hit (the "removeHit" method of TrackCandidate.java). If not, then the candidate is labeled not "good," and the code moves on to the next seed.

For good candidates, the code restarts the Kalman filter in "filterTrack" at the first layer, using as a starting guess the final result of the previous smoothing. When doing so it uses the existing hit assignments, but it can add hits on layers with no assignment. Some more candidates get labeled not good afterwards if the fit is bad or there ends up being too few hits. For the good candidates, the code then turns around and smooths back to the first layer. Resulting candidates that fail to intersect the target plane (*i.e.* way too much curvature) get labeled not good. For remaining candidates, there is then an attempt to improve them by removing "bad" hits, followed by a refit (the "reFit" method in TrackCandidate.java). Since some hit assignments can change, a check is made again to remove fully redundant candidates. A check is also made on the track-parameter covariance matrix, and any track with an NaN in the covariance gets marked no-good.

On the positive side, at this point if a track candidate is "golden," meaning that it has lots of hits and an excellent chi-squared, then it gets immediately removed from the candidate list and stored as a KalTrack object (using the "storeTrack" method), and its hits get marked in Measurement objects as being used. The used hits get removed from other candidates (which can result in them getting marked no-good, unless they satisfy the stringent requirements on hit sharing, and similarly they become unavailable to subsequent candidates.

If the candidate is not golden but is still good, then it remains in the list with a "good" designation, and its hits are marked in the KalHit objects (but not removed from other candidates).

Once all of the seeds have been exhausted for the given global iteration, then the resulting list of track candidates is evaluated. First, the bad candidates are removed, except that some have a chance to get "resurrected" and labeled good. Resurrection requires that the candidate share no hits with any other candidate or completed track, that it has a sufficient number of axial and stereo hits, and that it not have any NaN values in the helix parameters or covariance matrix. The idea here is to keep unique tracks even if they fit badly. The analysis can reject them later based on fit quality, but at least it won't look like they were invisible to the pattern recognition.

The list of good track candidates for the given global iteration is then sorted by quality, and a loop is made over all of the KalHit objects. For each hit there is a list of candidates using the hit. The list is paired down to just the best candidate, unless stringent hit-sharing requirements are met. Hence in general the best candidates get priority on the hits, which get removed from poorer candidates. The poor candidates that lose too many hits to remain viable then get marked no-good, and finally there is a list of good candidates that are unique, in the sense of not sharing hits unless the hit really fits both candidates with excellent chi-squared contribution (and in any case a limited number of shared hits per track is allowed, to avoid duplicate tracks).

Note that this methodology prevents the first candidate formed from preferentially grabbing the best hits. Instead, candidates are allowed to try and use hits already used by other candidates, and the preferences get sorted out at the end of the global iteration, in the manner described here. The code just has to be careful to avoid finding repeatedly the same track, which is why redundancy checks are made at several levels.

The remaining good candidates get stored away as KalTrack objects. The candidate list is then cleared, and the next global iteration begins (if there is one).

Finally, when the global iterations are completed, the resulting list of KalTrack objects is sorted by quality, and another check for hit sharing is done (using the track lists in the Measurement.java objects), and shared hits are allocated to only the best track, unless the sharing meets the stringent requirements. Tracks again are eliminated if they end up with too few hits. Otherwise they get their final Kalman-Filter fit, using the "fit" method of KalTrack. First, however, there is an attempt to add hits on layers missing from the track, using the "addHits" method of KalTrack, which on fairly rare occasions is successful. If the fit quality is really lousy, the code also tries stripping off bad hits and refitting. If no significant improvement is found it will go back to the original hit select and pass the lousy track on to the output—it does not throw it away. The fit tracks also get extrapolated to the target region, at which point they are ready to be converted into HPS track objects by methods in KalmanInterface.java.

15 Interface to HPS Java

The interface to the hps-java reconstruction flow is by way of the driver KalmanPatRecDriver.java. This driver in turn makes calls to methods in KalmanInterface.java. A single instance of KalmanInterface is constructed for the entire job and is simply cleared after each event. Similarly, the geometry for the Kalman code (SiModule.java) is set up once by a call to the "createSiModules" method of KalmanInterface.java. The method "prepareTrackCollections" calls the "KalmanPatRec" method of KalmanInterface.java, which in turn calls KalmanPatRecHPS for each of the two SVT detectors, after first filling the Kalman Measurement.java instances with the hit information.

The driver then converts the Kalman tracks to GBL tracks using the KalmanInterface method "createTrack," which also adds the hits to the track and creates a TrackState at each measurement layer and at the extrapolated point on the ECAL. The tracks are then persisted in a collection named "KalmanFullTracks."