HPS Kalman Filter Documentation July 25, 2019

R.P. Johnson U.C. Santa Cruz

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 has been 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, ultimately the code does have to be tested on a realistic simulation, so there is already an interface to hps-java that was written mostly by Miriam. So far the interface only tests the Kalman fitting, by providing hits already selected by the existing pattern recognition and already fit by the GBL algorithm. Making an interface for the pattern recognition remains to be done.

2 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):

$$\begin{aligned} d_0 &\leftarrow d_\rho \\ \phi_0 &\leftarrow -\phi_0 \\ \Omega &\leftarrow -\kappa/\alpha \\ z_0 &\leftarrow -d_z \end{aligned} \tag{3}$$

$$\tan\lambda \leftarrow -\tan\lambda \tag{4}$$

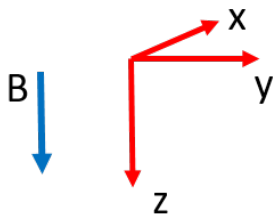
3 Coordinate Systems

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).

3.1 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:

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

3.2 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):

$$\begin{aligned} \hat{z}' &= \vec{B}/|\vec{B}| \\ \hat{x}' &= (\hat{y} \times \hat{z}')/|\hat{y} \times \hat{z}'| \\ \hat{y}' &= z' \times x' \end{aligned} \tag{6}$$

The origin of this local system is simply placed at the center of the detector. 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

$$\begin{aligned} \phi_0 &= \text{atan2}(-p_x, p_y) \\ \kappa &= Q/\sqrt{p_x^2 + p_y^2} \\ \tan\lambda &= p_z/\sqrt{p_x^2 + p_y^2} \end{aligned} \tag{7}$$

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

$$\frac{\partial a'}{\partial a} = \frac{\partial a'}{\partial p'} \cdot \frac{\partial p'}{\partial p} \cdot \frac{\partial p}{\partial a} \tag{8}$$

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

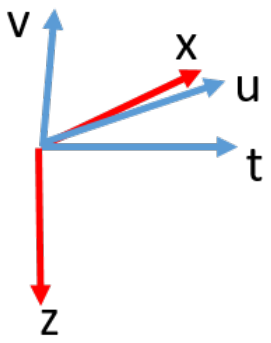
$$\frac{\partial p}{\partial a} = \begin{pmatrix} -\cos\phi_0/|\kappa| & \sin\phi_0/(\kappa \cdot |\kappa|) & 0 \\ -\sin\phi_0/|\kappa| & -\cos\phi_0/(\kappa \cdot |\kappa|) & 0 \\ 0 & -\tan\lambda/(\kappa \cdot |\kappa|) & 1/|\kappa| \end{pmatrix} \tag{9}$$

$$\frac{\partial a'}{\partial p'} = \begin{pmatrix} -p_y/p_t^2 & p_x/p_t^2 & 0 \\ -Qp_x/p_t^3 & -Qp_y/p_t^3 & 0 \\ -p_x p_z/p_t^3 & -p_y p_z/p_t^3 & 1/p_t \end{pmatrix} \tag{10}$$

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).

3.3 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 perfectly aligned sensor plane in a stereo layer with stereo angle θ:

$$R = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ -\sin\theta & 0 & -\cos\theta \\ 0 & 1 & 0 \end{pmatrix} \tag{11}$$

4 Track Propagation

4.1 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

$$f(\phi) \equiv (\vec{r}(\phi) - \vec{r}_0) \cdot \hat{t} \tag{12}$$

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.

4.2 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 StateVector.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). That normally amounts to propagating 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 5). 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 StateVector.java was written to do the transformation in multiple small helix steps, reorienting with the local field after each step. This method also includes multiple scattering of the first silicon layer in the covariance propagation. However, if the Kalman fit happened to end at a different silicon layer, because of missing hits in the first layer(s), the code does not yet add in the multiple scattering of all the intervening layers. *That coding job remains to be done.*

5 Pivot Transformation

A single helix is described by a pivot point and five helix parameters (see Section 2). 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

$$\begin{aligned} x_c =& x_0 + \left(d_\rho + \frac{\alpha}{\kappa}\right)\cos\phi_0 \\ y_c =& y_0 + \left(d_\rho + \frac{\alpha}{\kappa}\right)\sin\phi_0 \end{aligned} \tag{13}$$

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

$$\begin{aligned} d_\rho' =& (x_c - x_0')\cos\phi_0' + (y_c - y_0')\sin\phi_0' - \frac{\alpha}{\kappa} \\ \phi_0' =& \text{atan2}(y_c - y_0', x_c - x_0') \quad (\kappa > 0) \\ =& \text{atan2}(y_0' - y_c, x_0' - x_c) \quad (\kappa < 0) \\ \kappa' =& \kappa \\ d_z' =& z_0 - z_0' + d_z - \frac{\alpha}{\kappa}(\phi_0' - \phi_0)\tan\lambda \\ \tan\lambda' =& \tan\lambda \end{aligned} \tag{14}$$

In order to implement the Kalman filter, including propagation of the covariance matrix, the derivative matrix of the transformation 14 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

$$\begin{aligned} f_{00} =& \cos(\phi_0' - \phi_0) \\ f_{01} =& \left(d_\rho + \frac{\alpha}{\kappa}\right)\sin(\phi_0' - \phi_0) \\ f_{02} =& \left(\frac{\alpha}{\kappa^2}\right)\cdot(1 - \cos(\phi_0' - \phi_0)) \\ f_{10} =& -\frac{\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa} \\ f_{11} =& \frac{(d_\rho + \alpha/\kappa)\cos(\phi_0' - \phi_0)}{(d_\rho' + \alpha/\kappa)} \\ f_{12} =& \left(\frac{\alpha}{\kappa^2}\right)\frac{\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa} \\ f_{22} =& 1 \\ f_{30} =& \left(\frac{\alpha}{\kappa}\right)\frac{\tan\lambda\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa} \\ f_{31} =& \left(\frac{\alpha}{\kappa}\right)\tan\lambda\left(1 - \frac{(d_\rho + \alpha/\kappa)\cos(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa}\right) \\ f_{32} =& \left(\frac{\alpha}{\kappa^2}\right)\tan\lambda\left(\phi_0' - \phi_0 - \frac{(\alpha/\kappa)\sin(\phi_0' - \phi_0)}{d_\rho' + \alpha/\kappa}\right) \\ f_{33} =& 1 \\ f_{34} =& -\left(\frac{\alpha}{\kappa}\right)\cdot(\phi_0' - \phi_0) \\ f_{44} =& 1 \end{aligned} \tag{15}$$

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 3.2. 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. 15 to produce the complete transformation matrix.

6 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

$$\begin{aligned} q_{11} &= \left(1 + \tan^2\lambda\right) \cdot \sigma^2 \\ q_{44} &= \left(1 + \tan^2\lambda\right)^2 \cdot \sigma^2 \end{aligned} \tag{16}$$

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

$$\sigma = \frac{0.0136}{|p|}\sqrt{\frac{d}{X_0}} \cdot (1 + 0.038\ln(d/X_0)) \tag{17}$$

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.

7 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 8) 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.

7.1 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.

7.2 Prediction

The prediction step is done by the "makePrediction" method of MeasurementSite.java. It in turn calls the "predict" method of StateVector.java.

7.2.1 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 4.

- 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" $\mathbf{H}$ 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:

$$\sigma_r^2 = \sigma^2 - \sum_{ij} H_i C_{ij} H_j \tag{18}$$

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

$$\chi_+^2 = \frac{r^2}{\sigma_r^2} \tag{19}$$

7.2.2 The Derivatives H_i

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

$$H_i = \frac{\partial m}{\partial a_i} \tag{20}$$

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

$$\mathbf{M} = \mathbf{R}\mathbf{R}_f^{-1} \tag{21}$$

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

$$\frac{\partial m}{\partial a_j} = \sum_i M_{1j} \frac{\partial x_i}{\partial a_j} \tag{22}$$

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

$$\frac{\partial x_i(\phi(\mathbf{a}), \mathbf{a})}{\partial a_j} = \frac{\partial x_i}{\partial \phi} \frac{\partial \phi}{\partial a_j} + \frac{\partial x_i}{\partial a_j} \tag{23}$$

As discussed in Section 4.1, the point $\vec{x}$ is calculated from setting Eqn. 12 equal to zero. We can represent this as $f(\vec{x}(\phi(\mathbf{a}), \mathbf{a})) = 0$, which can be differentiated implicitly to find $\partial\phi/\partial a_j$:

$$\frac{\partial f}{\partial a_j} = \sum_i \frac{\partial f}{\partial x_i} \frac{\partial x_i(\phi(\mathbf{a}), \mathbf{a})}{\partial a_j} = \sum_i \hat{t}_i \left(\frac{\partial x_i}{\partial \phi} \frac{\partial \phi}{\partial a_j} + \frac{\partial x_i}{\partial a_j} \right) = 0 \tag{24}$$

From this it follows that

$$\frac{\partial \phi}{\partial a_j} = -\frac{\sum_i \hat{t}_i \frac{\partial x_i}{\partial a_j}}{\sum_i \hat{t}_i \frac{\partial x_i}{\partial \phi}} \tag{25}$$

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. 25 is substituted into Eqn. 23, which in turn is substituted into Eqn. 22.

The needed derivatives are

$$\begin{aligned} \frac{\partial x_0}{\partial \phi} &= \frac{\alpha}{\kappa} \sin(\phi_0 + \phi) \\ \frac{\partial x_1}{\partial \phi} &= -\frac{\alpha}{\kappa} \cos(\phi_0 + \phi) \\ \frac{\partial x_2}{\partial \phi} &= -\frac{\alpha}{\kappa} \tan\lambda \end{aligned} \tag{26}$$

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

$$\begin{aligned} \frac{\partial x_0}{\partial a_0} &= \cos\phi_0 \\ \frac{\partial x_1}{\partial a_0} &= \sin\phi_0 \\ \frac{\partial x_0}{\partial a_1} &= -\left(d_\rho + \frac{\alpha}{\kappa}\right) \sin\phi_0 + \frac{\alpha}{\kappa} \sin(\phi_0 + \phi) \\ \frac{\partial x_1}{\partial a_1} &= \left(d_\rho + \frac{\alpha}{\kappa}\right) \cos\phi_0 - \frac{\alpha}{\kappa} \cos(\phi_0 + \phi) \\ \frac{\partial x_0}{\partial a_2} &= -\frac{\alpha}{\kappa^2} (\cos\phi_0 - \cos(\phi_0 + \phi)) \\ \frac{\partial x_1}{\partial a_2} &= -\frac{\alpha}{\kappa^2} (\sin\phi_0 - \sin(\phi_0 + \phi)) \\ \frac{\partial x_2}{\partial a_2} &= \frac{\alpha}{\kappa^2} \tan\lambda \cdot \phi \\ \frac{\partial x_2}{\partial a_3} &= 1 \\ \frac{\partial x_2}{\partial a_4} &= -\frac{\alpha}{\kappa} \cdot \phi \end{aligned} \tag{27}$$

7.2.3 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. 14. 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 3.2. 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. 16, 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 $\mathbf{F}$:
$$\mathbf{C}' = \mathbf{F}^T\mathbf{C}\mathbf{F} \tag{28}$$

7.3 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. 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

$$K_i = \frac{\sum_j C_{ij}H_j}{\sigma^2 + \sum_{ij} H_i C_{ij} H_j} \tag{29}$$

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

The filtered helix parameters then are

$$a_{f_i} = a_i + K_i \cdot r \tag{30}$$

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

$$C_{f_{ij}} = \sum_m \left(I_{im} + K_i H_m\right) C_{mj} \tag{31}$$

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 18 and 19.

7.4 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

$$A_{ij} = \sum_{mn} C_{f\,im} F_{nm} {C'_p}^{-1}{}_{nj} \tag{32}$$

Then the smoothed helix parameters are

$$a_{si} = a_{f\,i} + \sum_j A_{ij} \left(a'_{sj} - a'_{p\,j} \right) \tag{33}$$

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

$$C_{sij} = C_{f\,ij} + \sum_{mn} A_{mi} \left(C'_{smn} - C'_{p\,mn} \right) A_{nj} \tag{34}$$

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.

8 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. 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 multiplied by a large factor before feeding it to the Kalman Filter routine. Normally the Kalman filter is iterated in a second pass, using the result of the first pass to initialize the second, again blowing up 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 I considered that to be a poor option. The 5-parameter linear fit is quite fast, anyway. The job is done by LinearHelixFit.java. For each of the N hits (typically 3 stereo and 2 axial) it 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

$$v_{\rm pred} = R_{10}\left[c + dy + ey^2 - \delta_0\right] + R_{11}\left[y - \delta_1\right] + R_{12}\left[a + by - \delta_2\right]. \tag{35}$$

The fit then minimizes the χ^2:

$$\chi^2 = \sum_{j=1}^{N} \left[\frac{v_{{\rm m}i} - v_{{\rm pred}_i}}{s_i}\right]^2, \tag{36}$$

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

$$\mathbf{A} = \sum_{i=1}^{N} \begin{pmatrix} w_i R_{12_i}^2 & w_i y_i R_{12_i}^2 & w_i R_{12i} R_{10i} & w_i y_i R_{12i} R_{10i} & w_i y_i^2 R_{12i} R_{10i} \\ w_i y_i R_{12_i}^2 & w_i y_i^2 R_{12i} R_{10i} & w_i y_i R_{12i} R_{10i} & w_i y_i^2 R_{12i} R_{10i} & w_i y_i^3 R_{12i} R_{10i} \\ w_i R_{12i} R_{10i} & w_i y_i R_{12i} R_{10i} & w_i R_{10_i}^2 & w_i y_i R_{10_i}^2 & w_i y_i^2 R_{10_i}^2 \\ w_i y_i R_{12i} R_{10i} & w_i y_i^2 R_{12i} R_{10i} & w_i y_i R_{10_i}^2 & w_i y_i^2 R_{10_i}^2 & w_i y_i^3 R_{10_i}^2 \\ w_i y_i^2 R_{12i} R_{10i} & w_i y_i^3 R_{12i} R_{10i} & w_i y_i^2 R_{10_i}^2 & w_i y_i^3 R_{10_i}^2 & w_i y_i^4 R_{10_i}^2 \end{pmatrix} \tag{37}$$

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

$$\vec{b} = \sum_{i=1}^{N} \begin{pmatrix} v_{ci} R_{12i} w_i \\ v_{ci} y_i R_{12i} w_i \\ v_{ci} R_{10i} w_i \\ v_{ci} y_i R_{10i} w_i \\ v_{ci} y_i^2 R_{10i} w_i \end{pmatrix} \tag{38}$$

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

$$\begin{aligned} x_c &= R \cdot d \\ y_c &= c + R\left(1 - d^2/2\right) \end{aligned} \tag{39}$$

The helix parameters, then, are

$$\begin{aligned} \phi_0 &= \text{atan2}\left(y_c, x_c\right) \\ \kappa &= \frac{\alpha}{R} \\ d_\rho &= \frac{x_c}{\cos\phi_0} - R \\ \tan\lambda &= a \cdot \cos\phi_0 \\ d_z &= b + d_\rho \tan\lambda \tan\phi_0 \end{aligned} \tag{40}$$

Some derivatives for transforming the convariance are

$$\begin{aligned} \frac{d\phi_0}{dc} &= \frac{2e}{d}\sin^2\phi_0 \\ \frac{d\phi_0}{dd} &= \frac{\sin\phi_0\cos\phi_0}{d - \sin^2\phi_0} \\ \frac{d\phi_0}{de} &= \frac{2c}{d}\sin^2\phi_0 \end{aligned} \tag{41}$$

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

$$C = D^T C_f D \tag{42}$$

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

$$
\begin{aligned}
D_{02} &= \frac{1}{\cos\phi_0} + x_c \frac{\tan\phi_0}{\cos\phi_0} \cdot \frac{d\phi_0}{dc} \\
D_{03} &= -\left(\frac{d}{2e\cos\phi_0}\right) + x_c \frac{\tan\phi_0}{\cos\phi_0} \cdot \frac{d\phi_0}{dd} \\
D_{04} &= \left[1 - \left(\frac{1 - d^2/2}{\cos\phi_0}\right)\right] / (2e^2) + x_c \frac{\tan\phi_0}{\cos\phi_0} \cdot \frac{d\phi_0}{de} \\
D_{12} &= \frac{d\phi_0}{dc} \\
D_{13} &= \frac{d\phi_0}{dd} \\
D_{14} &= \frac{d\phi_0}{de} \\
D_{24} &= 2\alpha \\
D_{30} &= 1 \\
D_{31} &= d_\rho \sin\phi_0 \\
D_{41} &= \cos\phi_0
\end{aligned}
\tag{43}
\tag{44}
$$

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.

9 Kalman Filter Refit of an Existing Track

The class KalmanTrackFit2 does a refit of a given track without doing any pattern recognition (i.e. not changing any hit assignments). Before calling it 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 7.1. 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 8, 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 send 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.

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

[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.

described in Section 4. 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.

10 Pattern Recognition using the Kalman Filter

The pattern recognition code is still in a preliminary state and has been tested thus far only with the stand-alone idealized simulation. It is contained in the class KalmanPatRecHPS.java. Its output is an array-list of KalTrack objects, one for each found track. The 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 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 makes 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 last part of the initialization is to set a bunch of parameters used by the algorithm. They are hardwired into the code up to now, but at least they are all collected together in one spot.

The pattern recognition is then contained within two nested loops. First there is a loop over iterations (so far set 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. Second there is a loop over the seed strategies.

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. For high occupancy this can get to be a lot of combinations, so some more thought might be put into how to quickly avoid uninteresting combinations. For now, at least, the code calls SeedTrack (see Section 8) for each combination, to do a linear 5-parameter fit to the 5 hits (i.e. zero degrees of freedom). 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.

The seeds that pass the cuts then get sorted for quality, and starting with the best seed, the Kalman filter is used to try to build a full 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 iteration. If the results of that filtering are acceptable, then it 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 it looks like the result is redundant with a set of hits already tried. Otherwise it restarts the Kalman filter at the first layer, using as a starting guess the final result of the smoothing. When doing so it uses the existing hit assignments, but it can add hits on layers with no assignment. When it gets to the last layer it turns around and smooths back to the first layer. While doing so, it throws out hits with unacceptably large contributions to the track χ^2, and on those layers it looks for a better hit to substitute.

The resulting track candidates are put into two lists of KalTrack objects, the good track candidates and the marginal ones. The hits on the good tracks are marked as used, but not those on the marginal tracks.

After fitting all the good seeds, the marginal tracks are then moved to the list of good tracks only if they do not share too many hits. Then all of the good tracks are reviewed, and shared hits are removed from all but the closest track unless they are very close to multiple tracks. Finally, the Kalman filter and smoothing is repeated for the good tracks.