Extended Kalman Filter

Keisuke Fujii

The ACFA-Sim-J Group

ABSTRACT

This short review stems from the appendix of Kotoyo Hoshina's Ph.D thesis and chapter 4 of Yasuhiro Nakashima's Master's thesis both written in Japanese. It is intended to be an introduction for those who are interested in principle of Kalman Filter and its application to track fitting in high energy collider experiments.

Contents

Chapter 1

Introduction

The ultimate goal of a detector system at a future linear e^+e^- collider experiment is to measure two 4-vectors, p^μ (4-momentum), and s^μ (spin polarization), of "partons" (quarks, leptons, gauge bosons, and other fundamental particles such as Higgs bosons, if any) produced in e^+e^- collisions so that we can visualize the underlying Feynman amplitudes and determine their corresponding S-matrix element. When you have quarks or gluons in the final states, however, what you actually observe is not these fundamental partons but jets of particles, which are predominantly stable hadrons. In order to reconstruct parent partons from the jets of particles, we need a high resolution particle tracker capable of efficient and accurate reconstruction of densely packed tracks of charge particles in the jets.

Track reconstruction consists of two steps: track finding, which selects hits belonging to a single track from the set of hits created by all the charged particles in an event, and track fitting, which fits the selected hits to a track model and determines its track parameters at the interaction point. The purpose of this short review is to elaborate this second step, track fitting, so as to find out the optimal way to extract track parameters at the interaction point.

In a uniform magnetic field, a charged particle follows a helical trajectory, which can be specified by 5 parameters: $(d_\rho, \phi_0, \kappa, d_z, \tan\lambda)$, if there is no multiple scattering or energy loss in detector materials. The simplest way to extract the track parameters is then to perform a χ^2 fit assuming that the particle follows a perfect helix, ignoring multiple scattering and energy loss. In this method, however, we have a single track parameter vector for a single set of hits. In the case of a low momentum tracks, for which multiple scattering and energy loss due to ionization are significant, a relatively strong disturbance in the middle of the flight would distort the trajectory and subsequent hits would deteriorates the fit near the interaction point (see Fig. 1.1-a)).

In order to solve this problem, HEP community started to use Kalman filter for track reconstruction in around 1987 [2, 3, 4]. The idea of Kalman filter first appeared in 1960 [1] to deal with a linear system under the influence of random disturbance. It is designed to effectively estimate, from multiple observations about a system, its state at a certain point. The method is effective in particular when

- equation of motion is (quasi-)linear[1],
- there are many discrete points of measurements,
- the distance between adjacent measurement points is short enough so that the state at k-th point is

[1]Since the original Kalman filter is for a linear system, its application to a non-linear system such as a charged particle trajectory in a magnetic field requires proper linearization. Strictly speaking, therefore, the Kalman filter we are going to discuss in what follows should be called an extended Kalman filter. We shall, however, call it Kalman filter in this note.

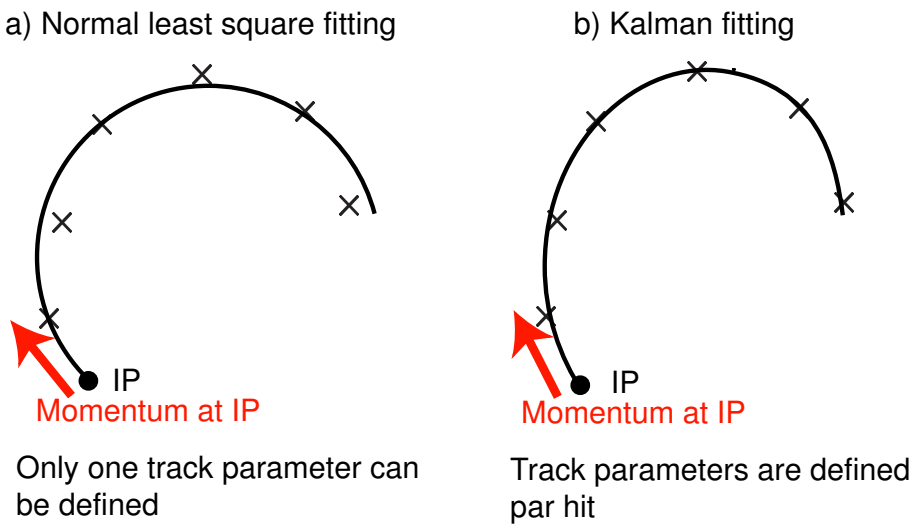

Figure 1.1: Simple χ^2 fitting versus Kalman filter based fitting.

well approximated by the extrapolation of the state at $(k-1)$-th point using the equation of motion, even in the presence of random disturbance between them.

The most crucial difference from the simple χ^2 fitting to a single helix is that we can put a different track parameter vector to a different measurement point, since we update the track parameters as we add a new measurement point. Because of this we can take into account step-by-step the evolution of the track parameters due to energy loss, multiple Coulomb scattering, and decays such as $K \to \mu\nu$ or $\pi \to \mu\nu$, etc., and hence improve the fit near the interaction point and consequently the estimation of the particle's momentum thereat (see Fig. 1.1-b)).

Chapter 2

Principle of Kalman Filter

2.1 Statement of the Problem

In general a system we deal with using Kalman filter is subject to random disturbance (*process noise*) during its evolution following an equation of motion (*system equation*). Our goal is to derive the best estimate of this system's state at a given point from information collected at multiple observation points (*measurement sites*).

We assume that the state of the system can be specified by a p-dimensional column vector (*state vector*), $\bar{\boldsymbol{a}}_k$, where the suffix k denotes the site at which the state is given and the bar over $\boldsymbol{a}$ means that it is the true value. The system equation that describes the evolution of the state $\bar{\boldsymbol{a}}_{k-1}$ at site $(k-1)$ to $\bar{\boldsymbol{a}}_k$ at site (k) can be written in the following form:

$$\bar{\boldsymbol{a}}_k = \boldsymbol{f}_{k-1}(\bar{\boldsymbol{a}}_{k-1}) + \boldsymbol{w}_{k-1}, \tag{2.1}$$

where $\boldsymbol{f}_{k-1}(\bar{\boldsymbol{a}}_{k-1})$, which is a non-linear function in general, is a *state propagator* corresponding to a smooth deterministic motion expected if the random disturbance due to the *process noise*, $\boldsymbol{w}_{k-1}$, were absent. We assume that the process noise has no bias: $< \boldsymbol{w}_{k-1} >= 0$ and has a covariance given by

$$\boldsymbol{Q}_{k-1} \equiv < \boldsymbol{w}_{k-1}\boldsymbol{w}_{k-1}^T > . \tag{2.2}$$

At each measurement site, we measure observables about the system. The set of observed values of these observables at site (k) forms an m-dimensional column vector called *measurement vector*, $\boldsymbol{m}_k$. The relation between the measurement vector and the state vector thereat is a *measurement equation*:

$$\boldsymbol{m}_k = \boldsymbol{h}_k(\bar{\boldsymbol{a}}_k) + \boldsymbol{\epsilon}_k, \tag{2.3}$$

where $\boldsymbol{h}_k(\bar{\boldsymbol{a}}_k)$ gives the true measurement vector that would be observed if the random error due to the *measurement noise*, $\boldsymbol{\epsilon}_k$, were absent. Notice that $\boldsymbol{h}_k(\bar{\boldsymbol{a}}_k)$ is also non-linear in general and serves as a *projector* that projects out components of the measurement vector. We again assume that the random measurement noise is unbiased[1]: $< \boldsymbol{\epsilon}_k >= 0$ and has a covariance given by

$$\boldsymbol{V}_k \equiv (\boldsymbol{G})^{-1} \equiv < \boldsymbol{\epsilon}_k\boldsymbol{\epsilon}_k^T > . \tag{2.4}$$

[1] In the case of tracking, this corresponds to assuming that the alignment is perfect and there is no systematic errors in coordinate measurements.

2.1.1 Example 1: Ballistic Missile

The Kalman filter was originally invented to track a ballistic missile and predict its trajectory. In this case, the state vector, $\boldsymbol{a}_k$, consists of position and momentum of the missile at site (k):

$$\boldsymbol{a}_k = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{p} \end{pmatrix}_k \tag{2.5}$$

which evolves according to Hamiltonian equation with the process noise, $\boldsymbol{w}_{k-1}$, corresponding to random turbulence between $(k-1)$ and (k).

The radar-detected position and velocity of the missile at site (k) comprises the measurement vector, $\boldsymbol{m}_k$, which is subject to a finite measurement error, $\boldsymbol{\epsilon}_k$, of the radar system.

2.1.2 Example 2: Tracking in HEP Experiments

Our main concern is tracking of a charged particle in a magnetic field. If the magnetic field is uniform, at least locally, near the measurement site (k), then we can approximate the track of this particle by a helix[5]:

$$\left\{ \begin{array}{rclcll} x & = & x_0 & + & d_\rho \cos\phi_0 & + & \frac{\alpha}{\kappa}\left(\cos\phi_0 - \cos(\phi_0+\phi)\right) \\ y & = & y_0 & + & d_\rho \sin\phi_0 & + & \frac{\alpha}{\kappa}\left(\sin\phi_0 - \sin(\phi_0+\phi)\right) \\ z & = & z_0 & + & d_z & - & \frac{\alpha}{\kappa}\tan\lambda\cdot\phi, \end{array} \right. \tag{2.6}$$

where $\boldsymbol{x}_0 = (x_0, y_0, z_0)_k^T$ is a pivotal point or simply pivot that represents a measurement site (k). At a given site, or equivalently for a given pivot, the helix is then specified by 5 parameters:

$$\boldsymbol{a}_k = \begin{pmatrix} d_\rho \\ \phi_0 \\ \kappa \\ d_z \\ \tan\lambda \end{pmatrix}_k . \tag{2.7}$$

This parameter vector serves as the state vector in this case. Notice that the change of pivot transforms the helix parameter vector. As we move from site $(k-1)$ to site (k), the state vector hence propagates from $\boldsymbol{a}_{k-1}$ to $\boldsymbol{a}_k$ according to the pivot transformation. The pivot transformation thus determines the function $\boldsymbol{f}_{k-1}$ in the system equation. On the other hand, the process noise, $\boldsymbol{w}_{k-1}$, in this case comes from multiple Coulomb scattering and energy loss between site $(k-1)$ and site (k). We shall elaborate these points later. The measurement vector $\boldsymbol{m}_k$ is a set of coordinates measured by a position sensitive detector at site (k), which is subject to random detector noise expressed by $\boldsymbol{\epsilon}_k$.

2.1.3 What We Need

Accumulation of the measurement vectors improves the estimate of the state of the system. What we need is hence machineary (recurrence formulae) to do

- *Prediction*, which predicts the state vector at site $(k'' > k)$ based on the observations made at sites up to (k):
$$\{\boldsymbol{m}'_k; k' \le k\} \mapsto \boldsymbol{a}^k_{k''>k} \qquad : \text{future}$$
- *Filtering*, which updates the predicted state vector at site (k) based on the observations made at sites up to $(k-1)$ by adding the observation at site (k):
$$\{\boldsymbol{m}'_k; k' \le k\} \mapsto \boldsymbol{a}^k_{k''=k} \qquad : \text{present}$$

- *Smoothing*, which improves the filtered state vector at site $(k'' < k)$ by using all the observations made at sites up to (k):
$$\{\boldsymbol{m}'_k; k' \leq k\} \mapsto \boldsymbol{a}^k_{k''<k} \qquad : \text{past}$$

In the rest of this chapter, we will derive recurrence formula for each of these three cases[2].

2.2 Prediction: Extrapolation to Next Site

We begin with the prediction, which predicts the state vector at site (k) from the measurements made at sites up to $(k-1)$. Let us recall that Eq.(2.1) relates the state vector at site $(k-1)$ to that at site (k). Notice that before making observation at site (k) we don't know anything about the random process noise (the second term on the right-hand side of Eq.(2.1)) that would disturb the smooth motion given by the first term of Eq.(2.1) as the system moves from site $(k-1)$ to (k). The best we can do is, hence, to extrapolate the state vector at site $(k-1)$ by using only the first term or the state propagator:

$$\boldsymbol{a}^{k-1}_k = \boldsymbol{f}_{k-1}(\boldsymbol{a}^{k-1}_{k-1}) = \boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1}). \tag{2.9}$$

By definition, the covariance matrix (error matrix) for $\boldsymbol{a}_{k-1} \equiv \boldsymbol{a}^{k-1}_{k-1}$ is given by

$$\begin{aligned} \boldsymbol{C}_{k-1} &\equiv \boldsymbol{C}^{k-1}_{k-1} \\ &\equiv \left\langle (\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1}) (\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1})^T \right\rangle . \end{aligned} \tag{2.10}$$

Similarly we can define the covariance matrix for the predicted state vector $(\boldsymbol{a}^{k-1}_k)$ and express it in terms of $\boldsymbol{C}_{k-1}$ with the help of Eqs.(2.9), (2.1), and (2.10):

$$\boldsymbol{C}^{k-1}_k \equiv \left\langle \left(\boldsymbol{a}^{k-1}_k - \bar{\boldsymbol{a}}_k\right) \left(\boldsymbol{a}^{k-1}_k - \bar{\boldsymbol{a}}_k\right)^T \right\rangle \tag{2.11}$$

$$\begin{aligned} &= \left\langle (\boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1}) - \boldsymbol{f}_{k-1}(\bar{\boldsymbol{a}}_k) - \boldsymbol{w}_{k-1}) (\boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1}) - \boldsymbol{f}_{k-1}(\bar{\boldsymbol{a}}_k) - \boldsymbol{w}_{k-1})^T \right\rangle \\ &\simeq \left\langle (\boldsymbol{F}_{k-1}(\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1}) - \boldsymbol{w}_{k-1}) (\boldsymbol{F}_{k-1}(\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1}) - \boldsymbol{w}_{k-1})^T \right\rangle \\ &= \boldsymbol{F}_{k-1} \left\langle (\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1}) (\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1})^T \right\rangle \boldsymbol{F}^T_{k-1} + \left\langle \boldsymbol{w}_{k-1} \boldsymbol{w}^T_{k-1} \right\rangle \\ &\quad + \text{cross terms} \\ &= \boldsymbol{F}_{k-1} \boldsymbol{C}_{k-1} \boldsymbol{F}^T_{k-1} + \left\langle \boldsymbol{w}_{k-1} \boldsymbol{w}^T_{k-1} \right\rangle \\ &\quad + \text{cross terms}, \end{aligned} \tag{2.12}$$

[2]The notation we shall use from now on can be summarized as follows.

$$\left\{ \begin{array}{lcl} \boldsymbol{a}^i_k & : & \text{estimate of } \bar{\boldsymbol{a}}_k \text{ using measurements at sites up to } (i) \\ & & \quad (\boldsymbol{a}^k_k \equiv \boldsymbol{a}_k \text{ for simplicity of notation}) \\ \boldsymbol{C}^i_k & : & \text{covariant matrix for } \boldsymbol{a}^i_k \\ & & \quad \boldsymbol{C}^i_k \equiv \left\langle (\boldsymbol{a}^i_k - \bar{\boldsymbol{a}}_k)(\boldsymbol{a}^i_k - \bar{\boldsymbol{a}}_k)^T \right\rangle \\ \boldsymbol{r}^i_k & : & \text{residual} \\ & & \quad \boldsymbol{r}^i_k \equiv \boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}^i_k) \\ \boldsymbol{R}^i_k & : & \text{covariance matrix for } \boldsymbol{r}^i_k \\ & & \quad \boldsymbol{R}^i_k \equiv \left\langle \boldsymbol{r}^i_k \boldsymbol{r}^{iT}_k \right\rangle \end{array} \right. \tag{2.8}$$

Notice that unless otherwise stated, suffix k means that it is at site (k) and superfix i means that it is from observations at sites up to (i). When there are two superffices like i, j, it means that it is from observations at sites (i) to (j). Notice also that when superfix coincides with suffix we shall often omit the superfix unless there is some fear for confusion.

where use has been made of

$$\begin{aligned} \boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1}) - \boldsymbol{f}_{k-1}(\bar{\boldsymbol{a}}_k) &\simeq \left(\frac{\partial \boldsymbol{f}_{k-1}}{\partial \boldsymbol{a}_{k-1}}\right)(\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1}) \\ &= \boldsymbol{F}_{k-1}(\boldsymbol{a}_{k-1} - \bar{\boldsymbol{a}}_{k-1}). \end{aligned}$$

Since the random process noise and the random fluctuation of the estimated state vector at site $(k-1)$ are statistically independent, the cross terms in the above equation vanish and because of Eq.(2.2), we arrive at

$$\boldsymbol{C}_k^{k-1} = \boldsymbol{F}_{k-1}\boldsymbol{C}_{k-1}\boldsymbol{F}_{k-1}^T + \boldsymbol{Q}_{k-1}, \tag{2.13}$$

where

$$\boldsymbol{F}_{k-1} \equiv \left(\frac{\partial \boldsymbol{f}_{k-1}}{\partial \boldsymbol{a}_{k-1}}\right) \tag{2.14}$$

is, hereafter, called a *propagator matrix*.

2.3 Filtering: Optimal Estimate at Current Site

As mentioned above, what we want to do in the filtering step is to update the predicted state vector at site (k) based on the observations at sites up to $(k-1)$ by including the observation at site (k) and make the optimal estimate of the state vector at site (k) possible with all the information so far collected at all the sites up to site (k). The goal here is to formulate this as a recurrence formula.

The fact that the covariance matrix for the predicted state vector $(\boldsymbol{a}_k^{k-1})$ is given by Eq.(2.13) implies that all we can say about the state vector at site (k) from what we have observed at sites up to $(k-1)$ can be summarized into the following single χ^2:

$$(\chi^2)_k^{k-1} = (\chi^2)_{k-1}^{k-1} + (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1})^T \left(\boldsymbol{C}_k^{k-1}\right)^{-1} (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1}),$$

where $\boldsymbol{a}_k^*$ is a new estimate of the state vector at site (k) to be optimized below by adding the information from the measurement at site (k), and $(\chi^2)_{k-1}^{k-1} \equiv (\chi^2)_{k-1}$ is the χ^2 up to site $(k-1)$ and is independent of $\boldsymbol{a}_k^*$.

Now we add the information from the new measurement made at site (k):

$$(\chi^2)_k^{k,k} = (\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^*))^T \boldsymbol{G} (\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^*)).$$

We then obtain the following as an increment to $(\chi^2)_{k-1}^{k-1}$:

$$\begin{aligned} \chi_+^2 &= (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1})^T \left(\boldsymbol{C}_k^{k-1}\right)^{-1} (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1}) \\ &\quad + (\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^*))^T \boldsymbol{G} (\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^*)) \qquad (2.15) \\ &= (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1})^T \left(\boldsymbol{C}_k^{k-1}\right)^{-1} (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1}) \\ &\quad + \left(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) - (\boldsymbol{h}_k(\boldsymbol{a}_k^*) - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}))\right)^T \boldsymbol{G} \left(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) - (\boldsymbol{h}_k(\boldsymbol{a}_k^*) - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}))\right) \\ &\simeq (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1})^T \left(\boldsymbol{C}_k^{k-1}\right)^{-1} (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1}) \\ &\quad + \left(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) - \boldsymbol{H}_k(\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1})\right)^T \boldsymbol{G} \left(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) - \boldsymbol{H}_k(\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1})\right), \qquad (2.16) \end{aligned}$$

where we assumed that $\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1}$ is small and retained up to the first order term in Taylor expansion:

$$\boldsymbol{h}_k(\boldsymbol{a}_k^*) \simeq \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) + \boldsymbol{H}_k\ (\boldsymbol{a}_k^* - \boldsymbol{a}_k^{k-1}),$$

where the derivative of the projector:

$$\boldsymbol{H}_k = \left(\frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{a}_k^{k-1}}\right) \tag{2.17}$$

is, hereafter, called a *projector matrix*. Since $(\chi^2)_{k-1}^{k-1}$ attains its minimum at $\boldsymbol{a}_{k-1}$ and is independent of $\boldsymbol{a}_k^*$[3], the optimal estimate of the state vector at site (k) ($\boldsymbol{a}_k \equiv \boldsymbol{a}_k^k$) we can get from observations made at sites up to (k) is the $\boldsymbol{a}_k^*$ that minimizes this increment (χ_+^2).

By solving the condition of extremum:

$$\frac{\partial \chi_+^2}{\partial \boldsymbol{a}_k^*} = 0$$

for $\boldsymbol{a}_k^*$, we readily obtain

$$\boldsymbol{a}_k = \boldsymbol{a}_k^{k-1} + \left[\left(\boldsymbol{C}_k^{k-1}\right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k\right]^{-1} \boldsymbol{H}_k^T \boldsymbol{G}_k \left(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1})\right)$$

and hence arrive at

$$\boldsymbol{a}_k = \boldsymbol{a}_k^{k-1} + \boldsymbol{K}_k \left(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1})\right), \tag{2.18}$$

where

$$\boldsymbol{K}_k = \left[\left(\boldsymbol{C}_k^{k-1}\right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k\right]^{-1} \boldsymbol{H}_k^T \boldsymbol{G}_k. \tag{2.19}$$

The matrix $\boldsymbol{K}_k$ defined here is called *Kalman Gain Matrix* and represents how the new measurement at site (k) improves the predicted state vector at site (k) (the second term of Eq.(2.18) corresponds to a new pull towards the measurement vector $\boldsymbol{m}_k$).

The above expression for the Kalman gain matrix can be cast into another form. Namely, from Eq.(2.19), we have

$$\begin{aligned}
\boldsymbol{K}_k \left(\boldsymbol{H}_k \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T + (\boldsymbol{G}_k)^{-1}\right) &= \left[\left(\boldsymbol{C}_k^{k-1}\right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k\right]^{-1} \left(\boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T + \boldsymbol{H}_k^T\right) \\
&= \left[\left(\boldsymbol{C}_k^{k-1}\right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k\right]^{-1} \\
&\quad \times \left[\left\{\boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k + \left(\boldsymbol{C}_k^{k-1}\right)^{-1} - \left(\boldsymbol{C}_k^{k-1}\right)^{-1}\right\} \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T + \boldsymbol{H}_k^T\right] \\
&= \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T
\end{aligned}$$

and hence

$$\boldsymbol{K}_k = \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T \left(\boldsymbol{V}_k + \boldsymbol{H}_k \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T\right)^{-1}, \tag{2.20}$$

where use has been made of Eq.(2.4).

With Eq.(2.18) together with either Eq.(2.19) or Eq.(2.20), we can update the predicted state vector $\boldsymbol{a}_k^{k-1}$ by including the information from the new measurement at site (k) and obtain the filtered state vector $\boldsymbol{a}_k \equiv \boldsymbol{a}_k^k$.

[3] We shall discuss how to update the estimate of the state vector at a past point later.

Let us now derive a recurrence formula to update the covariance matrix ($\boldsymbol{C}_k^{k-1}$) so as to obtain the covariance matrix for $\boldsymbol{a}_k$. By definition of covariance matrix together with Eq.(2.18), we have

$$\begin{aligned}
\boldsymbol{C}_k &\equiv \left\langle (\boldsymbol{a}_k - \bar{\boldsymbol{a}}_k)(\boldsymbol{a}_k - \bar{\boldsymbol{a}}_k)^T \right\rangle \\
&= \left\langle \left(\boldsymbol{a}_k^{k-1} + \boldsymbol{K}_k(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1})) - \bar{\boldsymbol{a}}_k \right) \left(\boldsymbol{a}_k^{k-1} + \boldsymbol{K}_k(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1})) - \bar{\boldsymbol{a}}_k \right)^T \right\rangle \\
&= \left\langle \left[(\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k) + \boldsymbol{K}_k \left\{ \boldsymbol{m}_k - \boldsymbol{h}_k(\bar{\boldsymbol{a}}_k) - \left(\boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) - \boldsymbol{h}_k(\bar{\boldsymbol{a}}_k) \right) \right\} \right] [\cdots]^T \right\rangle .
\end{aligned}$$

Substituting Eq.(2.3), we then obtain

$$\begin{aligned}
\boldsymbol{C}_k &= \left\langle \left[(\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k) + \boldsymbol{K}_k \left\{ \boldsymbol{\epsilon}_k - \left(\boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) - \boldsymbol{h}_k(\bar{\boldsymbol{a}}_k) \right) \right\} \right] [\cdots]^T \right\rangle \\
&\simeq \left\langle \left[(\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k) + \boldsymbol{K}_k \left\{ \boldsymbol{\epsilon}_k - \boldsymbol{H}_k(\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k) \right\} \right] [\cdots]^T \right\rangle \\
&= \left\langle \left[(1 - \boldsymbol{K}_k \boldsymbol{H}_k)(\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k) + \boldsymbol{K}_k \boldsymbol{\epsilon}_k \right] [\cdots]^T \right\rangle ,
\end{aligned}$$

where again we assumed that $\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k$ is small and hence retained only up to the first order term of Taylor expansion as in Eq.(2.17).

Since the difference between the filtered and true state vectors at site (k), $\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k$, is statistically independent of the measurement error at site (k), $\boldsymbol{\epsilon}_k$, their cross terms will vanish upon averaging. We thus obtain

$$\begin{aligned}
\boldsymbol{C}_k &= (1 - \boldsymbol{K}_k \boldsymbol{H}_k) \left\langle (\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k)(\boldsymbol{a}_k^{k-1} - \bar{\boldsymbol{a}}_k)^T \right\rangle (1 - \boldsymbol{K}_k \boldsymbol{H}_k)^T + \boldsymbol{K}_k \left\langle \boldsymbol{\epsilon}_k \boldsymbol{\epsilon}_k^T \right\rangle \boldsymbol{K}_k^T \\
&= (1 - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{C}_k^{k-1} (1 - \boldsymbol{K}_k \boldsymbol{H}_k)^T + \boldsymbol{K}_k \boldsymbol{V}_k \boldsymbol{K}_k^T \qquad (2.21) \\
&= (1 - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{C}_k^{k-1} - \left[(1 - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T - \boldsymbol{K}_k \boldsymbol{V}_k \right] \boldsymbol{K}_k^T \\
&= (1 - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{C}_k^{k-1} - \left[\boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T - \boldsymbol{K}_k (\boldsymbol{H}_k \boldsymbol{C}_k^{k-1} \boldsymbol{H}_k^T + \boldsymbol{V}_k) \right] \boldsymbol{K}_k^T \\
&= (1 - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{C}_k^{k-1}, \qquad (2.22)
\end{aligned}$$

where use has been made of Eqs.(2.11), (2.4), and (2.20).

Using Eq.(2.19), we can further reform the above formula for $\boldsymbol{C}_k$ as follows:

$$\begin{aligned}
\boldsymbol{C}_k &= (1 - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{C}_k^{k-1} \\
&= \left(1 - \left[\left(\boldsymbol{C}_k^{k-1} \right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \right]^{-1} \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \right) \boldsymbol{C}_k^{k-1} \\
&= \left[\left(\boldsymbol{C}_k^{k-1} \right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \right]^{-1} \left\{ \left(\boldsymbol{C}_k^{k-1} \right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k - \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \right\} \boldsymbol{C}_k^{k-1} \\
&= \left[\left(\boldsymbol{C}_k^{k-1} \right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \right]^{-1} .
\end{aligned}$$

We thus arrive at

$$\boldsymbol{C}_k = \left[\left(\boldsymbol{C}_k^{k-1} \right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \right]^{-1} , \qquad (2.23)$$

which gives the filtered covariance matrix as weighted mean. This formula indicates that the new covariance matrix, $\boldsymbol{C}_k$, has indeed smaller eigen values than the predicted one, $\boldsymbol{C}_k^{k-1}$. Combining this formula and Eq.(2.19), we obtain yet another formula for the Kalman Gain Matrix:

$$\boldsymbol{K}_k = \boldsymbol{C}_k \boldsymbol{H}_k^T \boldsymbol{G}_k . \qquad (2.24)$$

2.4 Smoothing: Reevaluation at Past Site

In general, the state vector obtained at site (k) in the filtering process is the optimal at that point. The state vector at site (k) can, however, be reevaluated and further improved after we accumulate more information at subsequent sites: $(k+1)$ to (n). This process is called *Smoothing*. Obviously the result of filtering at the last site, (n), coincides that of smoothing. What we need to know is therefore how to smooth the filtered state vector site by site, starting from site (n) and moving back toward site (1). Our goal is hence to work out an inverse recurrence formula that gives the smoothed state vector at site (k) in terms of the smoothed state vector at site $(k+1)$ and the filtered state vectors at sites (k) and $(k+1)$.

For this purpose, we start with the smoothing at site $(k+1)$. The optimal estimate of the state vector $(\boldsymbol{a}_{k+1}^n)$ at site $(k+1)$ using the information from all the n sites should be given as the weighted mean of the predicted state vector $(\boldsymbol{a}_{k+1}^k)$ at site $(k+1)$ using observations made at sites (1) to (k) and the backward filtered state vector $(\boldsymbol{a}_{k+1}^{n,k+1})$ based on observations made at sites (n) to $(k+1)$. This weighted mean is obtained by minimizing the following χ^2:

$$\begin{aligned}\chi^2 &= (\boldsymbol{a}_{k+1}^n - \boldsymbol{a}_{k+1}^k)^T \left(\boldsymbol{C}_{k+1}^k\right)^{-1} (\boldsymbol{a}_{k+1}^n - \boldsymbol{a}_{k+1}^k) \\ &\quad +(\boldsymbol{a}_{k+1}^n - \boldsymbol{a}_{k+1}^{n,k+1})^T \left(\boldsymbol{C}_{k+1}^{n,k+1}\right)^{-1} (\boldsymbol{a}_{k+1}^n - \boldsymbol{a}_{k+1}^{n,k+1}).\end{aligned}$$

Differentiating this χ^2 with respect to $\boldsymbol{a}_{k+1}^n$ and setting the result to zero, we readily obtain the smoothed state vector at site $(k+1)$:

$$\boldsymbol{a}_{k+1}^n = \left[\left(\boldsymbol{C}_{k+1}^k\right)^{-1} + \left(\boldsymbol{C}_{k+1}^{n,k+1}\right)^{-1}\right]^{-1} \left[\left(\boldsymbol{C}_{k+1}^k\right)^{-1} \boldsymbol{a}_{k+1}^k + \left(\boldsymbol{C}_{k+1}^{n,k+1}\right)^{-1} \boldsymbol{a}_{k+1}^{n,k+1}\right].$$

On the other hand, its covariance matrix is given by

$$(\boldsymbol{C}_{k+1}^n)^{-1} = \left(\boldsymbol{C}_{k+1}^k\right)^{-1} + \left(\boldsymbol{C}_{k+1}^{n,k+1}\right)^{-1}.$$

Solving the above two equations for $\boldsymbol{a}_{k+1}^{n,k+1}$ and $\left(\boldsymbol{C}_{k+1}^{n,k+1}\right)^{-1}$, we have

$$\boldsymbol{a}_{k+1}^{n,k+1} = \boldsymbol{a}_{k+1}^n + \boldsymbol{C}_{k+1}^n \left[\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n\right]^{-1} (\boldsymbol{a}_{k+1}^n - \boldsymbol{a}_{k+1}^k) \tag{2.25}$$

$$\left(\boldsymbol{C}_{k+1}^{n,k+1}\right)^{-1} = (\boldsymbol{C}_{k+1}^n)^{-1} - \left(\boldsymbol{C}_{k+1}^k\right)^{-1}. \tag{2.26}$$

Similarly, for the smoothed state vector at site (k) we obtain

$$\boldsymbol{a}_k^n = \left[(\boldsymbol{C}_k)^{-1} + \left(\boldsymbol{C}_k^{n,k+1}\right)^{-1}\right]^{-1} \left[(\boldsymbol{C}_k)^{-1} \boldsymbol{a}_k + \left(\boldsymbol{C}_k^{n,k+1}\right)^{-1} \boldsymbol{a}_k^{n,k+1}\right] \tag{2.27}$$

$$(\boldsymbol{C}_k^n)^{-1} = (\boldsymbol{C}_k)^{-1} + \left(\boldsymbol{C}_k^{n,k+1}\right)^{-1}. \tag{2.28}$$

Our goal is now to eliminate factors with superfix $n, k+1$ from the above two equations, making use of Eqs.(2.25) and (2.26).

From the definition of covariance matrix:

$$\boldsymbol{C}_k^{n,k+1} \equiv \left\langle (\boldsymbol{a}_k^{n,k+1} - \bar{\boldsymbol{a}}_k)(\boldsymbol{a}_k^{n,k+1} - \bar{\boldsymbol{a}}_k)^T \right\rangle$$

and Eq.(2.1), we obtain

$$\boldsymbol{C}_k^{n,k+1} = \left\langle \left(\boldsymbol{f}_k^{-1}(\boldsymbol{a}_{k+1}^{n,k+1}) - \boldsymbol{f}_k^{-1}(\bar{\boldsymbol{a}}_{k+1} - \boldsymbol{w}_k)\right) \left(\boldsymbol{f}_k^{-1}(\boldsymbol{a}_{k+1}^{n,k+1}) - \boldsymbol{f}_k^{-1}(\bar{\boldsymbol{a}}_{k+1} - \boldsymbol{w}_k)\right)^T \right\rangle$$

$$
\begin{aligned}
&\simeq \left\langle \left(\boldsymbol{F}_k^{-1}(\boldsymbol{a}_{k+1}^{n,k+1} - \bar{\boldsymbol{a}}_{k+1} + \boldsymbol{w}_k) \right) \left(\boldsymbol{F}_k^{-1}(\boldsymbol{a}_{k+1}^{n,k+1} - \bar{\boldsymbol{a}}_{k+1} + \boldsymbol{w}_k) \right)^T \right\rangle \\
&= \boldsymbol{F}_k^{-1} \left\langle \left((\boldsymbol{a}_{k+1}^{n,k+1} - \bar{\boldsymbol{a}}_{k+1}) + \boldsymbol{w}_k \right) \left((\boldsymbol{a}_{k+1}^{n,k+1} - \bar{\boldsymbol{a}}_{k+1}) + \boldsymbol{w}_k \right)^T \right\rangle \boldsymbol{F}_k^{-1T},
\end{aligned}
$$

where we assumed that $\boldsymbol{a}_{k+1}^{n,k+1}$ and $\bar{\boldsymbol{a}}_{k+1} - \boldsymbol{w}_k$ are close and hence we can approximate $\boldsymbol{f}_k^{-1}$ to a good accuracy by its Taylor expansion to the first order. Noting that the process noise between site (k) to site $(k+1)$ is statistically independent of the difference, $\boldsymbol{a}_{k+1}^{n,k+1} - \bar{\boldsymbol{a}}_{k+1}$, and making use of Eq.(2.2) and the definition of covariance matrix, we obtain

$$
\boldsymbol{C}_k^{n,k+1} = \boldsymbol{F}_k^{-1} \left(\boldsymbol{C}_{k+1}^{n,k+1} + \boldsymbol{Q}_k \right) \boldsymbol{F}_k^{-1T}. \tag{2.29}
$$

Since $\boldsymbol{C}_{k+1}^{n,k+1}$ is given by Eq.(2.26), we have thus eliminated factors with superfix $n, k+1$.

On the other hand, Eqs.(2.27) and (2.28) lead us to

$$
\begin{aligned}
\boldsymbol{a}_k^n &= \boldsymbol{C}_k^n \left\{ (\boldsymbol{C}_k^n)^{-1} \boldsymbol{a}_k + \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} (\boldsymbol{a}_k^{n,k+1} - \boldsymbol{a}_k) \right\} \\
&= \boldsymbol{a}_k + \boldsymbol{C}_k^n \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} (\boldsymbol{a}_k^{n,k+1} - \boldsymbol{a}_k) \\
&= \boldsymbol{a}_k + \boldsymbol{C}_k^n \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} \left(\boldsymbol{f}_k^{-1}(\boldsymbol{a}_{k+1}^{n,k+1}) - \boldsymbol{f}_k^{-1}(\boldsymbol{a}_{k+1}^k) \right) \\
&\simeq \boldsymbol{a}_k + \boldsymbol{C}_k^n \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} \boldsymbol{F}_k^{-1}(\boldsymbol{a}_{k+1}^{n,k+1} - \boldsymbol{a}_{k+1}^k),
\end{aligned}
$$

resulting in

$$
\boldsymbol{a}_k^n = \boldsymbol{a}_k + \boldsymbol{A}_k(\boldsymbol{a}_{k+1}^n - \boldsymbol{a}_{k+1}^k) \tag{2.30}
$$

$$
\boldsymbol{A}_k \equiv \boldsymbol{C}_k^n \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} \boldsymbol{F}_k^{-1} \boldsymbol{C}_{k+1}^k \left(\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n \right)^{-1} \tag{2.31}
$$

after some simple arithmetics with the help of Eq.(2.25). $\boldsymbol{C}_k^n$ and $\boldsymbol{C}_k^{n,k+1}$ in $\boldsymbol{A}_k$ can be eliminated by using Eq.(2.28) and Eqs.(2.29) and (2.26), respectively, as follows:

$$
\begin{aligned}
\boldsymbol{A}_k &\equiv \boldsymbol{C}_k^n \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} \boldsymbol{F}_k^{-1} \boldsymbol{C}_{k+1}^k \left(\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n \right)^{-1} \\
&= \left[(\boldsymbol{C}_k)^{-1} + \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} \right]^{-1} \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} \boldsymbol{F}_k^{-1} \boldsymbol{C}_{k+1}^k \left(\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n \right)^{-1} \\
&= \left[\boldsymbol{C}_k^{n,k+1} \left\{ (\boldsymbol{C}_k)^{-1} + \left(\boldsymbol{C}_k^{n,k+1} \right)^{-1} \right\} \right]^{-1} \boldsymbol{F}_k^{-1} \boldsymbol{C}_{k+1}^k \left(\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n \right)^{-1} \\
&= \left[\left(\boldsymbol{C}_k + \boldsymbol{C}_k^{n,k+1} \right) (\boldsymbol{C}_k)^{-1} \right]^{-1} \boldsymbol{F}_k^{-1} \boldsymbol{C}_{k+1}^k \left(\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n \right)^{-1} \\
&= \boldsymbol{C}_k \left(\boldsymbol{C}_k + \boldsymbol{C}_k^{n,k+1} \right)^{-1} \boldsymbol{F}_k^{-1} \boldsymbol{C}_{k+1}^k \left(\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n \right)^{-1}.
\end{aligned} \tag{2.32}
$$

On the other hand, Eqs.(2.13) and (2.29) imply

$$
\boldsymbol{C}_k + \boldsymbol{C}_k^{n,k+1} = \boldsymbol{F}_k^{-1} \left(\boldsymbol{C}_{k+1}^k + \boldsymbol{C}_{k+1}^{n,k+1} \right) \boldsymbol{F}_k^{-1T},
$$

which together with Eq.(2.26) leads us to

$$
\begin{aligned}
\boldsymbol{C}_k + \boldsymbol{C}_k^{n,k+1} &= \boldsymbol{F}_k^{-1} \left(\boldsymbol{C}_{k+1}^k + \boldsymbol{C}_{k+1}^{n,k+1} \right) \boldsymbol{F}_k^{-1T} \\
&= \boldsymbol{F}_k^{-1} \left(\boldsymbol{C}_{k+1}^k + \boldsymbol{C}_{k+1}^k \left(\boldsymbol{C}_{k+1}^k - \boldsymbol{C}_{k+1}^n \right)^{-1} \boldsymbol{C}_{k+1}^n \right) \boldsymbol{F}_k^{-1T}
\end{aligned}
$$

$$= \boldsymbol{F}_k^{-1}\boldsymbol{C}_{k+1}^k\left(\boldsymbol{C}_{k+1}^k-\boldsymbol{C}_{k+1}^n\right)^{-1}\boldsymbol{C}_{k+1}^k\boldsymbol{F}_k^{-1T}. \tag{2.33}$$

Substituting this in Eq.(2.32) and combining the result with Eq.(2.30), we finally arrive at

$$\begin{cases} \boldsymbol{a}_k^n &= \boldsymbol{a}_k+\boldsymbol{A}_k(\boldsymbol{a}_{k+1}^n-\boldsymbol{a}_{k+1}^k) \\ \boldsymbol{A}_k &= \boldsymbol{C}_k\boldsymbol{F}_k^T\left(\boldsymbol{C}_{k+1}^k\right)^{-1}, \end{cases} \tag{2.34}$$

which is none other than the recurrence formula for $\boldsymbol{a}_k^n$ we have been looking for.

For completeness let us now derive the covariance matrix for $\boldsymbol{a}_k^n$. From Eqs.(2.32) and (2.33), we obtain

$$\boldsymbol{C}_k^n\left(\boldsymbol{C}_k^{n,k+1}\right)^{-1} = \boldsymbol{C}_k\boldsymbol{F}_k^T\left(\boldsymbol{C}_{k+1}^k\right)^{-1}\left(\boldsymbol{C}_{k+1}^k-\boldsymbol{C}_{k+1}^n\right)\left(\boldsymbol{C}_{k+1}^k\right)^{-1}\boldsymbol{F}_k.$$

Multiplying the both sides by $\boldsymbol{C}_k^{n,k+1}$ from the right and substituting $\boldsymbol{C}_k^{n,k+1}$ with Eq.(2.33), we get

$$\boldsymbol{C}_k^n = \boldsymbol{C}_k-\boldsymbol{C}_k\boldsymbol{F}_k^T\left(\boldsymbol{C}_{k+1}^k\right)^{-1}\left(\boldsymbol{C}_{k+1}^k-\boldsymbol{C}_{k+1}^n\right)\left(\boldsymbol{C}_{k+1}^k\right)^{-1}\boldsymbol{F}_k\boldsymbol{C}_k,$$

resulting in the desired formula:

$$\boldsymbol{C}_k^n = \boldsymbol{C}_k+\boldsymbol{A}_k\left(\boldsymbol{C}_{k+1}^n-\boldsymbol{C}_{k+1}^k\right)\boldsymbol{A}_k^T, \tag{2.35}$$

where use has been made of Eq.(2.34).

2.5 Inverse Kalman Filter: Exclusion of Site

As our last exercise let us consider the case in which we want to exclude a site in the middle, say site (k), from the estimate of the state vector, after we perform Kalman filter from site (1) to site (n) and smooth back from site (n) to site (1). Of course, we can always redo Kalman filter as if there were no site (k) to eliminate the site. There is, however, better way that makes full use of the previous results with site (k) at hand. This process is called *Inverse Kalman Filter* and is useful, in particular, when we want to perform alignment of the measurement layer at site (k) as is often the case with tracking devices.

Elimination of site (k) corresponds to the following subtraction for χ^2:

$$\chi^{*2} = (\boldsymbol{a}_k^{n*}-\boldsymbol{a}_k^n)^T(\boldsymbol{C}_k^n)^{-1}(\boldsymbol{a}_k^{n*}-\boldsymbol{a}_k^n)-(\boldsymbol{m}_k-\boldsymbol{h}_k(\boldsymbol{a}_k^{n*})^T\boldsymbol{G}_k(\boldsymbol{m}_k-\boldsymbol{h}_k(\boldsymbol{a}_k^{n*}).$$

By differentiating this new χ^{*2} with respect to $\boldsymbol{a}_k^{n*}$ and setting the result to zero, we can write down an equation to derive the optimal estimate of the state vector at site (k) with the measurement vector at site (k) left out.

Comparison of this with Eq.(2.15) for the filtering process tells us that the following substitutions:

$$\begin{array}{rcl} \boldsymbol{a}_k^{k-1} & \rightarrow & \boldsymbol{a}_k^n \\ \boldsymbol{C}_k^{k-1} & \rightarrow & \boldsymbol{C}_k^n \\ \boldsymbol{G}_k & \rightarrow & -\boldsymbol{G}_k \end{array}$$

transforms χ_+^2 to χ^{*2}. We can thus readily translate the formulae for the filtering process, Eqs.(2.18), (2.20), and (2.23), to the formulae for the inverse Kalman filter:

$$\boldsymbol{a}_k^{n*} = \boldsymbol{a}_k^n+\boldsymbol{K}_k^{n*}(\boldsymbol{m}_k-\boldsymbol{h}_k(\boldsymbol{a}_k^n)) \tag{2.36}$$

$$\boldsymbol{K}_k^{n*} = \boldsymbol{C}_k^n\boldsymbol{H}_k^T\left(-\boldsymbol{V}_k+\boldsymbol{H}_k\boldsymbol{C}_k^n\boldsymbol{H}_k^T\right)^{-1} \tag{2.37}$$

$$\begin{aligned} \boldsymbol{C}_k^{n*} &= (1-\boldsymbol{K}_k^{n*}\boldsymbol{H}_k)\boldsymbol{C}_k\ n \\ &= \left[(\boldsymbol{C}_k^n)^{-1}-\boldsymbol{H}_k^T\boldsymbol{G}_k\boldsymbol{H}_k\right]^{-1}. \end{aligned} \tag{2.38}$$

Notice that the essential difference is the sign in front of $\boldsymbol{G}_k$ or $\boldsymbol{V}_k$, which indicates the negative weight associated with the eliminated information at site (k).

Chapter 3

Application of Kalman Filter to Track Fitting

3.1 Basic Formulae for Kalman Filter

In this chapter we formulate a procedure for Kalman-filter based track fitting in high energy collider experiments. A general purpose 4π detector at a collider experiment is usually equipped with a solenoidal magnet and in its axial magnetic field a tracking system cylindrically surrounding the interaction point at its center. The tracking system often consists of multiple measurement layers so as to sample locations of a particle along its trajectory.

Let us begin with picking out necessary formulae for the Kalman-filter based track fitting from the recurrence formulae we derived in the previous chapter. When we apply Kalman filter to track fitting, we usually start from a hit in the outermost measurement layer and move back toward the interaction point. This is because the average two-track distance is largest for the outermost layer and hence chance for hit point confusion is minimal. Moreover, when we reach the innermost hit point, the filtered state vector is already optimal and requires no smoothing. Unless there is need for track extrapolation to outer detectors, the recurrence formulae we need are the following two, which give the filtered state vector ($\boldsymbol{a}_k$) and its covariance matrix ($\boldsymbol{C}_k$) at site (k):

$$\begin{aligned} \boldsymbol{a}_k &= \boldsymbol{a}_k^{k-1} + \boldsymbol{K}_k \left(\boldsymbol{m}_k - \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1}) \right) \\ \boldsymbol{C}_k &= \left[\left(\boldsymbol{C}_k^{k-1} \right)^{-1} + \boldsymbol{H}_k^T \boldsymbol{G}_k \boldsymbol{H}_k \right]^{-1}, \end{aligned} \tag{3.1}$$

in terms of the predicted state vector ($\boldsymbol{a}_k^{k-1}$), the Kalman gain matrix ($\boldsymbol{K}_k$), the residual of the measurement vector ($\boldsymbol{m}_k$) with respect to the expected measurement vector ($\boldsymbol{h}_k(\boldsymbol{a}_k^{k-1})$), the covariance matrix ($\boldsymbol{C}_k^{k-1}$) for the predicted state vector, and the inverse of the measurement error matrix ($\boldsymbol{G}_k \equiv (\boldsymbol{V}_k)^{-1}$). Notice that the Kalman gain matrix is given by

$$\boldsymbol{K}_k = \boldsymbol{C}_k \boldsymbol{H}_k^T \boldsymbol{G}_k,$$

and the predicted state vector and its covariance matrix by

$$\begin{aligned} \boldsymbol{a}_k^{k-1} &= \boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1}) \\ \boldsymbol{C}_k^{k-1} &= \boldsymbol{F}_{k-1}\, \boldsymbol{C}_{k-1}\, \boldsymbol{F}_{k-1}^T + \boldsymbol{Q}_{k-1}, \end{aligned}$$

where $\boldsymbol{Q}_{k-1}$ is the covariance matrix for the process noise between site $(k-1)$ and site (k) and $\boldsymbol{F}_{k-1}$ and

$\boldsymbol{H}_k$ are two application-specific derivatives, the propagator matrix ($\boldsymbol{F}_{k-1}$) and the projector matrix ($\boldsymbol{H}_k$):

$$\begin{cases} \boldsymbol{F}_{k-1} & \equiv & \left(\frac{\partial \boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1})}{\partial \boldsymbol{a}_{k-1}}\right) \\ \boldsymbol{H}_k & \equiv & \left(\frac{\partial \boldsymbol{h}_k(\boldsymbol{a}_k^{k-1})}{\partial \boldsymbol{a}_k^{k-1}}\right), \end{cases}$$

which are derived from system and measurement equations.

These formulae allow us to update the state vector as we accumulate information on the system by including a new measurement vector at a new site on a site-by-site basis, once we have application specific functions such as $\boldsymbol{f}_{k-1}$, $\boldsymbol{h}_k$, and their derivatives, $\boldsymbol{F}_{k-1}$ and $\boldsymbol{H}_k$ together with concrete formulae for the process noise ($\boldsymbol{Q_{k-1}}$) and the measurement noise ($\boldsymbol{V}_k$).

3.2 System Equation: $\boldsymbol{f}_{k-1}$, $\boldsymbol{F}_{k-1}$, and $\boldsymbol{Q}_{k-1}$

Since the measurement equation is detector dependent, let us first consider the system equation for a charged particle track in a magnetic field and try to find out concrete formulae for $\boldsymbol{f}_{k-1}$, $\boldsymbol{F}_{k-1}$, and $\boldsymbol{Q}_{k-1}$. For this purpose we need to specify a track model.

As we mentioned earlier, if the magnetic field is uniform and in the direction of z-axis, at least locally, near the measurement sites in question, then we can approximate the track of this particle by a helix (see Eq.(2.6)):

$$\begin{cases} x &=& x_0 &+& d_\rho \cos\phi_0 &+& \frac{\alpha}{\kappa}\left(\cos\phi_0 - \cos(\phi_0+\phi)\right) \\ y &=& y_0 &+& d_\rho \sin\phi_0 &+& \frac{\alpha}{\kappa}\left(\sin\phi_0 - \sin(\phi_0+\phi)\right) \\ z &=& z_0 &+& d_z &-& \frac{\alpha}{\kappa}\tan\lambda\cdot\phi, \end{cases} \tag{3.2}$$

where $\boldsymbol{x}_0 = (x_0, y_0, z_0)^T_{k-1}$ is a pivotal point or simply pivot that represents a measurement site $(k-1)$ and ϕ is the deflection angle measured from the pivot. Since the pivot is an arbitrary reference point, we can take it to be the hit point at site $(k-1)$ or equivalently the $(k-1)$-th hit point. At a given site, or equivalently for a given pivot, the helix is then specified by 5 parameters:

$$\begin{cases} d_\rho & : & \text{the distance between the helix and the pivot in the } x\text{-}y \text{ plane} \\ \phi_0 & : & \text{the azimuthal angle of the pivot with respect to the center of the helix} \\ \kappa & : & \equiv Q/P_t : \text{the charge in units of that of proton / the transverse momentum} \\ d_z & : & \text{the distance between the helix and the pivot in the } z \text{ direction} \\ \tan\lambda & : & \text{the dip angle, i.e., the angle of the helix to the } x\text{-}y \text{ plane,} \end{cases} \tag{3.3}$$

which form a state vector $\boldsymbol{a}_{k-1} = (d_\rho, \phi_0, \kappa, d_z, \tan\lambda)^T$. The paramter κ is related to the signed radius ρ of the helix[1] through $\rho = \alpha/\kappa$, where α is a constant given in terms of the speed of light (c) and the magnetic field (B) as $\alpha \equiv 1/cB$.

The helix parameters are depicted in Fig.3.1. Notice that the definition of ϕ_0 differs by π, depending on the sign of the track charge. This is to avoid discontinuity of ϕ_0 that would take place when the curvature of the track changes its sign during track fitting, by compensating the sudden jump of the center of the helix from one side of the track to the other.

3.2.1 State Propagator: $\boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1})$

Notice that the change of pivot from site $(k-1)$ to site (k) transforms the helix parameter vector without changing the helix itself. As we move from site $(k-1)$ to site (k), the state vector hence propagates from $\boldsymbol{a}_{k-1}$

[1] It should be stressed that we should use κ in stead of ρ as a parameter for track fitting, since only then we can allow continuous sign change of curvature as needed in fitting a high momentum track.

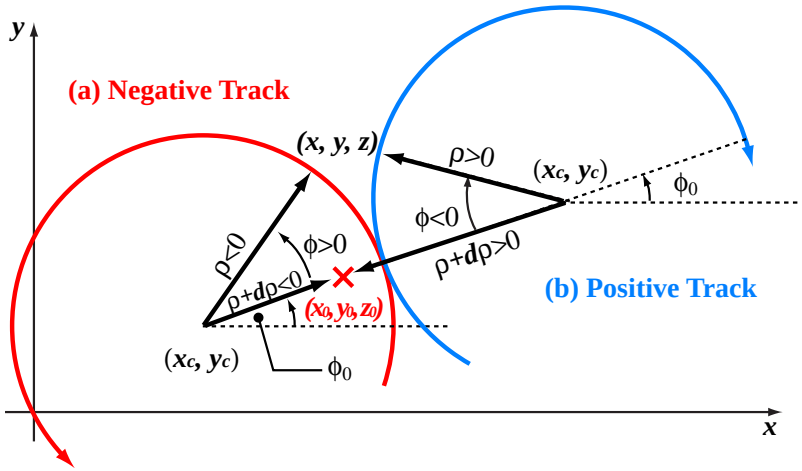

Figure 3.1: Geometric interpretation of track parameters for (a) negatively and (b) positively charged tracks.

to $\boldsymbol{a}_k^{k-1}$ according to the pivot transformation. The pivot transformation thus determines the propagator function $\boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1})$ in the system equation.

Fig.3.2 shows how the track parameters:

$$\boldsymbol{a}_{k-1} \equiv \boldsymbol{a} = (d_\rho, \phi_0, \kappa, d_z, \tan\lambda)^T$$

transform as we change pivot, where we denote a point on the helix by $\boldsymbol{x}$, a pivot (hit point) by $\boldsymbol{x}_0$, and the center of helix by $\boldsymbol{X}_c$. The primed parameters:

$$\boldsymbol{a}' \equiv \boldsymbol{a}_k^{k-1} = (d'_\rho, \phi'_0, \kappa', d'_z, \tan\lambda')^T = \boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1})$$

are those for the new pivot.

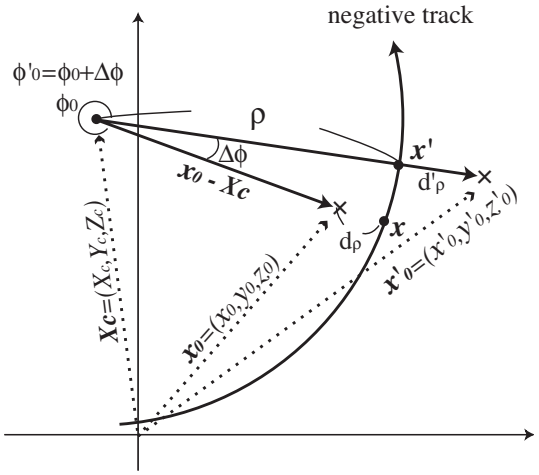

Figure 3.2: Relation between track parameters and pivot.

By inspection of Fig.3.2, we can read off the following relation between the primed and the unprimed

helix parameters:

$$\left\{\begin{array}{lcl} d'_\rho & = & (X_c - x'_0)\cos\phi'_0 + (Y_c - y'_0)\sin\phi'_0 - \frac{\alpha}{\kappa} \\ \phi'_0 & = & \left\{\begin{array}{ll} \tan^{-1}\left(\frac{Y_c - y'_0}{X_c - x'_0}\right) & (\kappa > 0) \\ \tan^{-1}\left(\frac{y'_0 - Y_c}{x'_0 - X_c}\right) & (\kappa < 0) \end{array}\right. \\ \kappa' & = & \kappa \\ d'_z & = & z_0 - z'_0 + d_z - \left(\frac{\alpha}{\kappa}\right)(\phi'_0 - \phi_0)\tan\lambda \\ \tan\lambda' & = & \tan\lambda, \end{array}\right. \tag{3.4}$$

where

$$\left\{\begin{array}{lcl} X_c & \equiv & x_0 + \left(d\rho + \frac{\alpha}{\kappa}\right)\cos\phi_0 \\ Y_c & \equiv & y_0 + \left(d\rho + \frac{\alpha}{\kappa}\right)\sin\phi_0\,. \end{array}\right. \tag{3.5}$$

Equation (3.4) together with Eq.(3.5) defines the propagator function $\boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1})$ for the state vector.

3.2.2 Propagator Matrix: $\boldsymbol{F}_{k-1}$

By differentiating Eq.(3.4) with respect to $\boldsymbol{a} = (d_\rho, \phi_0, \kappa, d_z, \tan\lambda)^T$ we can calculate the propagator matrix $\boldsymbol{F}_{k-1}$:

$$\boldsymbol{F}_{k-1} \equiv \left(\frac{\partial \boldsymbol{a}'}{\partial \boldsymbol{a}}\right) = \left(\begin{array}{c} \frac{\partial d'_\rho}{\partial \boldsymbol{a}} \\ \frac{\partial \phi'_0}{\partial \boldsymbol{a}} \\ \frac{\partial \kappa'}{\partial \boldsymbol{a}} \\ \frac{\partial d'_z}{\partial \boldsymbol{a}} \\ \frac{\partial \tan\lambda'}{\partial \boldsymbol{a}} \end{array}\right).$$

Notice that κ and $\tan\lambda$ do not change by the pivot transformation and hence the calculation of their partial derivatives is trivial. For the calculation of the partial derivatives of the rest we will start with ϕ'_0 since it appears also in d'_ρ and d'_z.

$\underline{\partial\phi'_0/\partial\boldsymbol{a}}$

Noting that the coordinates of the new pivot, x'_0, y'_0, and z'_0, are not parameters but arbitrarily chosen constants, we obtain from Eq.(3.4) the partial derivative of ϕ'_0 with respect to a variable ω to be

$$\frac{\partial\phi'_0}{\partial\omega} = \cos^2\phi'_0\left(\frac{\frac{\partial Y_c}{\partial\omega}}{X_c - x'_0} - \tan\phi'_0\frac{\frac{\partial X_c}{\partial\omega}}{X_c - x'_0}\right), \tag{3.6}$$

where (X_c, Y_c) is related to (x'_0, y'_0) through Eq.(3.5) as

$$\left\{\begin{array}{lcl} X_c & \equiv & x'_0 + \left(d'_\rho + \frac{\alpha}{\kappa'}\right)\cos\phi'_0 \\ Y_c & \equiv & y'_0 + \left(d'_\rho + \frac{\alpha}{\kappa'}\right)\sin\phi'_0\,, \end{array}\right.$$

resulting in

$$\left\{\begin{array}{lcl} X_c - x'_0 & = & \left(d'_\rho + \frac{\alpha}{\kappa}\right)\cos\phi'_0 \\ Y_c - y'_0 & = & \left(d'_\rho + \frac{\alpha}{\kappa}\right)\sin\phi'_0 \end{array}\right. \tag{3.7}$$

since $\kappa' = \kappa$. To complete the differentiation, all we need now is to calculate $\partial X_c/\partial \boldsymbol{a}$ and $\partial Y_c/\partial \boldsymbol{a}$, using Eq.(3.5):

$$\begin{cases} \dfrac{\partial X_c}{\partial d_\rho} &= \cos\phi_0 \\ \dfrac{\partial X_c}{\partial \phi_0} &= -\left(d'_\rho + \dfrac{\alpha}{\kappa}\right)\sin\phi_0 \\ \dfrac{\partial X_c}{\partial \kappa} &= -\dfrac{\alpha}{\kappa^2}\cos\phi_0 \\ \dfrac{\partial X_c}{\partial d_z} &= 0 \\ \dfrac{\partial X_c}{\partial \tan\lambda} &= 0 \end{cases} \qquad \begin{cases} \dfrac{\partial Y_c}{\partial d_\rho} &= \sin\phi_0 \\ \dfrac{\partial Y_c}{\partial \phi_0} &= \left(d'_\rho + \dfrac{\alpha}{\kappa}\right)\cos\phi_0 \\ \dfrac{\partial Y_c}{\partial \kappa} &= -\dfrac{\alpha}{\kappa^2}\sin\phi_0 \\ \dfrac{\partial Y_c}{\partial d_z} &= 0 \\ \dfrac{\partial Y_c}{\partial \tan\lambda} &= 0\,. \end{cases} \tag{3.8}$$

The desired partial derivatives are then readily obtained by substituting Eqs.(3.7) and (3.8) in Eq.(3.6) as needed. The resultant $\frac{\partial \phi'_0}{\partial \mathbf{a}}$ is given by

$$\begin{cases} \dfrac{\partial \phi'_0}{\partial d_\rho} &= -\left(d'_\rho + \dfrac{\alpha}{\kappa}\right)^{-1}\sin(\phi'_0 - \phi_0) \\ \dfrac{\partial \phi'_0}{\partial \phi_0} &= \left(d_\rho + \dfrac{\alpha}{\kappa}\right)\left(d'_\rho + \dfrac{\alpha}{\kappa}\right)^{-1}\cos(\phi'_0 - \phi_0) \\ \dfrac{\partial \phi'_0}{\partial \kappa} &= \dfrac{\alpha}{\kappa^2}\left(d'_\rho + \dfrac{\alpha}{\kappa}\right)^{-1}\sin(\phi'_0 - \phi_0) \\ \dfrac{\partial \phi'_0}{\partial d_z} &= 0 \\ \dfrac{\partial \phi'_0}{\partial \tan\lambda} &= 0\,, \end{cases} \tag{3.9}$$

regardless of the sign of κ.

$\underline{\partial d'_\rho/\partial \boldsymbol{a}}$

As with ϕ'_0, we obtain the following from Eq.(3.4) for the partial derivative of d'_ρ with respect to a variable ω:

$$\begin{aligned} \frac{\partial d'_\rho}{\partial \omega} &= \frac{\partial X_c}{\partial \omega}\cos\phi'_0 - (X_c - x'_0)\sin\phi'_0\frac{\partial \phi'_0}{\partial \omega} \\ &+ \frac{\partial Y_c}{\partial \omega}\sin\phi'_0 + (Y_c - y'_0)\cos\phi'_0\frac{\partial \phi'_0}{\partial \omega} - \frac{\partial}{\partial \omega}\frac{\alpha}{\kappa}\,. \end{aligned}$$

Using this equation together with Eqs.(3.8) and (3.9), we can carry out partial differentiation of d'_ρ with respect to $\boldsymbol{a}$ and obtain

$$\begin{cases} \dfrac{\partial d'_\rho}{\partial d_\rho} &= \cos(\phi'_0 - \phi_0) \\ \dfrac{\partial d'_\rho}{\partial \phi_0} &= \left(d_\rho + \dfrac{\alpha}{\kappa}\right) \sin(\phi'_0 - \phi_0) \\ \dfrac{\partial d'_\rho}{\partial \kappa} &= \dfrac{\alpha}{\kappa^2} \left(1 - \cos(\phi'_0 - \phi_0)\right) \\ \dfrac{\partial d'_\rho}{\partial d_z} &= 0 \\ \dfrac{\partial d'_\rho}{\partial \tan\lambda} &= 0 \end{cases} \tag{3.10}$$

regardless of the sign of κ.

Other Partial Derivatives

Partial derivatives of κ' and $\tan\lambda'$ are trivial:

$$\begin{cases} \dfrac{\partial \kappa'}{\partial d_\rho} &= \dfrac{\partial \kappa'}{\partial \phi_0} = \dfrac{\partial \kappa'}{\partial d_z} = \dfrac{\partial \kappa'}{\partial \tan\lambda} = 0 \\ \dfrac{\partial \kappa'}{\partial \kappa} &= 1 \\ \dfrac{\partial \tan\lambda'}{\partial d_\rho} &= \dfrac{\partial \tan\lambda'}{\partial \phi_0} = \dfrac{\partial \tan\lambda'}{\partial \kappa} = \dfrac{\partial \tan\lambda'}{\partial d_z} = 0 \\ \dfrac{\partial \tan\lambda'}{\partial \tan\lambda} &= 1. \end{cases} \tag{3.11}$$

With the help of Eq.(3.9) partial derivatives of d'_z are also easily obtained to be

$$\begin{cases} \dfrac{\partial d'_z}{\partial d_\rho} &= \dfrac{\alpha}{\kappa} \left(d'_\rho + \dfrac{\alpha}{\kappa}\right)^{-1} \tan\lambda \sin(\phi'_0 - \phi_0) \\ \dfrac{\partial d'_z}{\partial \phi_0} &= \dfrac{\alpha}{\kappa} \tan\lambda \left(1 - \left(d_\rho + \dfrac{\alpha}{\kappa}\right) \left(d'_\rho + \dfrac{\alpha}{\kappa}\right)^{-1} \cos(\phi'_0 - \phi_0)\right) \\ \dfrac{\partial d'_z}{\partial \kappa} &= \dfrac{\alpha}{\kappa^2} \tan\lambda \left(\phi'_0 - \phi_0 - \dfrac{\alpha}{\kappa} \left(d'_\rho + \dfrac{\alpha}{\kappa}\right)^{-1} \sin(\phi'_0 - \phi_0)\right) \\ \dfrac{\partial d'_z}{\partial d_z} &= 1 \\ \dfrac{\partial d'_z}{\partial \tan\lambda} &= -\dfrac{\alpha}{\kappa}(\phi'_0 - \phi_0) \end{cases} \tag{3.12}$$

regardless of the sign of κ.

3.2.3 Process Noise: $\boldsymbol{Q}_{k-1}$

For completeness, we discuss here how to implement multiple Coulomb scattering as the process noise, $\boldsymbol{w}_{k-1}$, between site $(k-1)$ and site (k).

Notice that in the recurrence formulae given by Eq.(3.1) the effect of the process noise only appears as its variance, $\boldsymbol{Q}_{k-1}$, in

$$\boldsymbol{C}_k^{k-1} \;=\; \boldsymbol{F}_{k-1}\,\boldsymbol{C}_{k-1}\,\boldsymbol{F}_{k-1}^T + \boldsymbol{Q}_{k-1}.$$

Of course, the variance of the process noise depends of the distribution of materials between site $(k-1)$ and site (k). For simplicity, let us first assume that the distance between the two sites is small enough so that we can approximate the multiple scattering to take place as if all the materials between site $(k-1)$ and site (k) are concentrated on an infinitely thin layer at the mid point $(m; k-1<m<k)$ between site $(k-1)$ and site (k). In this thin layer limit[5], the covariance matrix $\boldsymbol{Q}_m$ of the helix parameter vector at the mid point (m) takes the following simple form:

$$\boldsymbol{Q}_m \;=\; \sigma_{MS}^2 \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1+\tan^2\lambda & 0 & 0 & 0 \\ 0 & 0 & (\kappa\tan\lambda)^2 & 0 & \kappa\tan\lambda(1+\tan^2\lambda) \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \kappa\tan\lambda(1+\tan^2\lambda) & 0 & (1+\tan^2\lambda)^2 \end{pmatrix}, \tag{3.13}$$

where the κ and $\tan\lambda$ are those at the mid point but can be approximated by those at site $(k-1)$, provided that the distance between site $(k-1)$ and the mid point is small enough to do so, and σ_{MS} is given as usual by

$$\sigma_{MS} = \frac{0.0141}{P(\mathrm{GeV})\beta}\sqrt{X_L}\left(1+\frac{1}{9}\log_{10}X_L\right), \tag{3.14}$$

where P, β, and X_L are the momentum, the speed in units of the speed of light, and the thickness of the scatterer in units of its radiation length.

The desired covariance matrix for the process noise $\boldsymbol{Q}_{k-1}$ is then obtained by moving the pivot from the mid point (m) to site (k):

$$\boldsymbol{Q}_{k-1} \;=\; \boldsymbol{F}_{m,k}\,\boldsymbol{Q}_m\,\boldsymbol{F}_{m,k}^T, \tag{3.15}$$

where $\boldsymbol{F}_{m,k}$ is the propagator matrix for the covariance matrix[2] of the state vector from the mid point (m) to site (k) as given by Eqs.(3.6), (3.10), (3.11), and (3.12) with $\boldsymbol{a}=\boldsymbol{a}_m$ and $\boldsymbol{a}'=\boldsymbol{a}_{k-1}^k$.

When the track segment between site $(k-1)$ and site (k) is not short enough to apply the thin layer approximation elaborated above, we can subdivide the segment into N steps and perform the following summation:

$$\boldsymbol{Q}_{k-1} \;=\; \sum_{s=1}^{N-1} \boldsymbol{F}_{m_s,k}\,\boldsymbol{Q}_{m_s}\,\boldsymbol{F}_{m_s,k}^T. \tag{3.16}$$

3.3 Measurement Equation: $\boldsymbol{h}_k$, $\boldsymbol{H}_k$, and $\boldsymbol{V}_k$

Having derived the formulae related to the system equation, let us now move on to the derivations of the formulae related to the measurement equation:

$$\boldsymbol{m}_k \;=\; \boldsymbol{h}_k(\bar{\boldsymbol{a}}_k) + \boldsymbol{\epsilon}_k,$$

[2] In this notation $\boldsymbol{F}_{k-1} \equiv \boldsymbol{F}_{k-1,k}$

where the measurement vector $\boldsymbol{m}_k$ is a set of coordinates measured by a position sensitive detector at site (k), which is subject to random detector noise expressed by $\boldsymbol{\epsilon}_k$.

What we need is the projector $(\boldsymbol{h}_k(\boldsymbol{a}))$, which gives the exact coordinates of the hit point at site (k) as a function of the true track parameter vector at site (k), the projector matrix $(\boldsymbol{H}_k)$, which is its derivative with respect to the track parameter vector, and the covariance matrix $(\boldsymbol{V}_k)$ for the measurement noise.

In practice, locations of a charged particle are sampled at discrete points along its trajectory. We can imagine layers of (virtual) surfaces on which we sample the locations and call them measurement surfaces or layers. Each measurement layer corresponds to a measurement site, say site (k), and records the coordinates of its intersection, a hit point, with the particle's trajectory. The coordinates of the hit point comprise a measurement vector $\boldsymbol{m}_k$.

3.3.1 Projector: $\boldsymbol{h}_k(\boldsymbol{a})$

In order to predict the measurement vector on a measurement layer as a function of the state vector, i.e., the track parameter vector, we need, first of all, an equation that gives the intersection, $\boldsymbol{x}_k(\boldsymbol{a})$, of a track specified by $\boldsymbol{a}$ with the measurement surface given by $S_k(\boldsymbol{x}) = 0$:

$$\boldsymbol{x}_k \;=\; \boldsymbol{x}_k(\boldsymbol{a}) \;=\; \boldsymbol{x}(\phi_k(\boldsymbol{a}), \boldsymbol{a}) \tag{3.17}$$

and another equation that translates the intersection $\boldsymbol{x}_k(\boldsymbol{a})$ to the measurement vector:

$$\boldsymbol{m}_k = \boldsymbol{m}_k(\boldsymbol{x}_k)\,, \tag{3.18}$$

where $\boldsymbol{x} = \boldsymbol{x}(\phi, \boldsymbol{a})$ is a trajectory equation, which is given by Eq.(3.2) in the case of a helical track, and $\phi_k(\boldsymbol{a})$ is the deflection angle at which the track hits the measurement surface corresponding to site (k).

Given these two equations, we can now define the projector $\boldsymbol{h}_k(\boldsymbol{a})$ by

$$\begin{aligned} \boldsymbol{h}_k(\boldsymbol{a}) &= \boldsymbol{m}_k(\boldsymbol{x}_k(\boldsymbol{a})) \\ &= \boldsymbol{m}_k(\boldsymbol{x}(\phi_k(\boldsymbol{a}), \boldsymbol{a}))\,. \end{aligned} \tag{3.19}$$

3.3.2 Projector Matrix: $\boldsymbol{H}_k$

Although concrete forms of these equations depend on the nature of the measurement layer and hence application-specific, we can write down a general formula for the derivative of $\boldsymbol{h}_k$ with respect to $\boldsymbol{a}$:

$$\boldsymbol{H}_k \equiv \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{a}} = \left(\frac{\partial \boldsymbol{m}_k}{\partial \boldsymbol{x}}\right)\left(\frac{\partial \boldsymbol{x}(\phi_k(\boldsymbol{a}), \boldsymbol{a})}{\partial \boldsymbol{a}}\right), \tag{3.20}$$

where the second factor on the right-hand side is given by

$$\frac{\partial \boldsymbol{x}(\phi_k(\boldsymbol{a}), \boldsymbol{a})}{\partial \boldsymbol{a}} = \frac{\partial \boldsymbol{x}}{\partial \phi_k}\frac{\partial \phi_k}{\partial \boldsymbol{a}} + \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{a}}\,, \tag{3.21}$$

in which $\partial \boldsymbol{x}/\partial \phi_k$ and $\partial \boldsymbol{x}/\partial \boldsymbol{a}$ are determined solely by the trajectory equation. On the other hand, $\partial \boldsymbol{m}_k/\partial \boldsymbol{x}$ is determined by the geometry of the measurement surface $S_k(\boldsymbol{x}) = 0$ and the layout of the coordinate system on it, which are both application-specific.

Let us hence calculate $\partial \phi_k/\partial \boldsymbol{a}$ below. We start from the equation for ϕ_k:

$$S_k(\boldsymbol{x}(\phi_k, \boldsymbol{a})) = 0. \tag{3.22}$$

Differentiating both sides by the track parameter vector, we obtain

$$0 = \frac{\partial S_k}{\partial \boldsymbol{a}} = \frac{\partial S_k}{\partial \boldsymbol{x}}\left(\frac{\partial \boldsymbol{x}}{\partial \phi_k}\frac{\partial \phi_k}{\partial \boldsymbol{a}} + \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{a}}\right), \tag{3.23}$$

from which we have

$$\frac{\partial \phi_k}{\partial \boldsymbol{a}} = -\left(\frac{\partial S_k}{\partial \boldsymbol{x}}\right)\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{a}}\right) \Big/ \left(\frac{\partial S_k}{\partial \boldsymbol{x}}\right)\left(\frac{\partial \boldsymbol{x}}{\partial \phi_k}\right) . \tag{3.24}$$

Notice that $\partial S_k/\partial \boldsymbol{x}$ is determined solely by the geometry of the measurement surface and $\partial \boldsymbol{x}/\partial \phi$ and $\partial \boldsymbol{x}/\partial \boldsymbol{a}$ by the trajectory equation alone.

Whether Eq.(3.22) has an analytic solution or not depends on the geometry of the measurement surface $S_k(\boldsymbol{x}) = 0$ and it is often difficult to write down the analytic solution in a compact form even if it exists at all, unless we are dealing with particularly simple surfaces like cylinders of planes. In such a case, we rely on Newtonian method to solve the equation. Assume that we have an approximate solution ϕ_n, then we can Taylor-expand the equation as

$$0 = S_k(\boldsymbol{x}(\phi, \boldsymbol{a})) \;\simeq\; S_k(\boldsymbol{x}(\phi_n, \boldsymbol{a})) + \frac{\partial S_k}{\partial \boldsymbol{x}} \cdot \frac{\partial \boldsymbol{x}}{\partial \phi_n} \cdot (\phi - \phi_n) , \tag{3.25}$$

which results in a recurrence formula:

$$\phi_{n+1} = \phi_n - \frac{S_k(\boldsymbol{x}(\phi_n, \boldsymbol{a}))}{\left(\frac{\partial S_k}{\partial \boldsymbol{x}}\right)_n \cdot \left(\frac{\partial \boldsymbol{x}}{\partial \phi}\right)_n} . \tag{3.26}$$

We can use this recurrence formula iteratively until ϕ_n converges and hence $S_k(\boldsymbol{x}(\phi_n, \boldsymbol{a}))$ becomes negligibly small[3]. If the measurement layer in question is near the current pivot, we can usually start from $\phi_1 = 0$ and get the iteration converged after a few iterations.

3.3.3 Measurement Noise: $\boldsymbol{V}_k$

To complete the necessary set of the formulae for Kalman filter, we need the covariance matrix for the measurement noise ($\boldsymbol{V}_k$), which is in our case given by an error matrix that represents the resolution of the position sensitive detector at site (k). Denoting the residual measurement vector by

$$\Delta \boldsymbol{m}_k \;\equiv\; \boldsymbol{m}_k - \boldsymbol{h}_k(\bar{\boldsymbol{a}}_k) ,$$

we can define the covariance matrix by

$$\boldsymbol{V}_k \equiv \left\langle \Delta \boldsymbol{m}_k \; \Delta \boldsymbol{m}_k^T \right\rangle . \tag{3.27}$$

If we have two coordinates measured on the measurement surface at site (k) and if the two measured coordinates are statistically independent, we can write the covariance matrix in the form:

$$\boldsymbol{V}_k \;=\; \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix} , \tag{3.28}$$

where σ_1 and σ_2 are the resolutions for the two coordinates, respectively. Although these resolutions depend on the hit location and the track angle, etc. in general, we can take $\boldsymbol{V}_k$ as a constant matrix in the track fitting.

[3] In practice it is necessary to introduce a damping factor $(1 + \Lambda)$ to the denominator of the second term of Eq.(3.26), where we start from $\Lambda \ll 1$ and if $|S_k|$ increases we multiply Λ by a factor greater than 1 and if $|S_k|$ decreases we divide it by the same factor.

Chapter 4

Implementation in C++

4.1 Design Philosophy

Since the tracking system of a collider detector usually consists of various components such as a vertex detector (VD), an intermediate tracker (IT), a central tracker (CT), etc., which have different shapes and coordinate systems, the software package for Kalman-filter-based track fitting should be able to accommodate a measurement layer with any shape and/or coordinate system. Considering possible extrapolation of a track to an outer tracking device such as a muon detector, it is also desirable that the package can handle site-to-site change of the magnetic field. In order to satisfy these requirements with minimum user-implemented code, we shall use C++ and try to exploit its object-oriented features as much as possible. For persistency of data, we shall rely on ROOT's automatic schema evolution.

Notice that the Kalman filter defines a generic procedure and has a much wider scope than track fitting. This suggests necessity for a library of generic abstract base classes (KalLib) that implement the generic algorithm of the Kalman Filter. By inheriting from the generic base classes in KalLib and implementing their pure virtual methods for track fitting purpose, we can then realize a Kalman-filter-based track fitter library (KalTrackLib). However, KalTrackLib should not depend on any particular track model or shape or coordinate system of a measurement layer according to the above guideline. We hence separate out geometry classes that provide track model (helix, straight line, ...) and surfaces (cylinder, hyperboloid, flat plane, etc.) as a geometry library (GeomLib). Each of these three class libraries, KalLib, KalTrackLib, and GeomLib are distributed as a subpackage of LEDA (Library Extension for Data Analysis) or KalTest (Kalman filter Test bench).

In this way we can minimize the number of user-implemented classes to the following three:

- MeasLayer: a measurement layer that multiply inherits from an abstract measurement layer class `TVMeasLayer` and a shape class in GeomLib.
- KalDetector: an array that holds user-defined MeasLayers with any shape and/or coordinate system. Notice that this also defines material distributions in the tracker.
- Hit: a coordinate vector as defined by the MeasLayer.

Notice also that track model can change site to site, which allows magnetic field variation along a particle trajectory.

4.2 KalLib: Base Class Library for Kalman Filter

Let us start with implementation of the base class library (KalLib) to describe a generic Kalman filter process.

Fig.4.1 is a class diagram showing the architecture of KalLib. The `TVKalSystem` class is an abstract

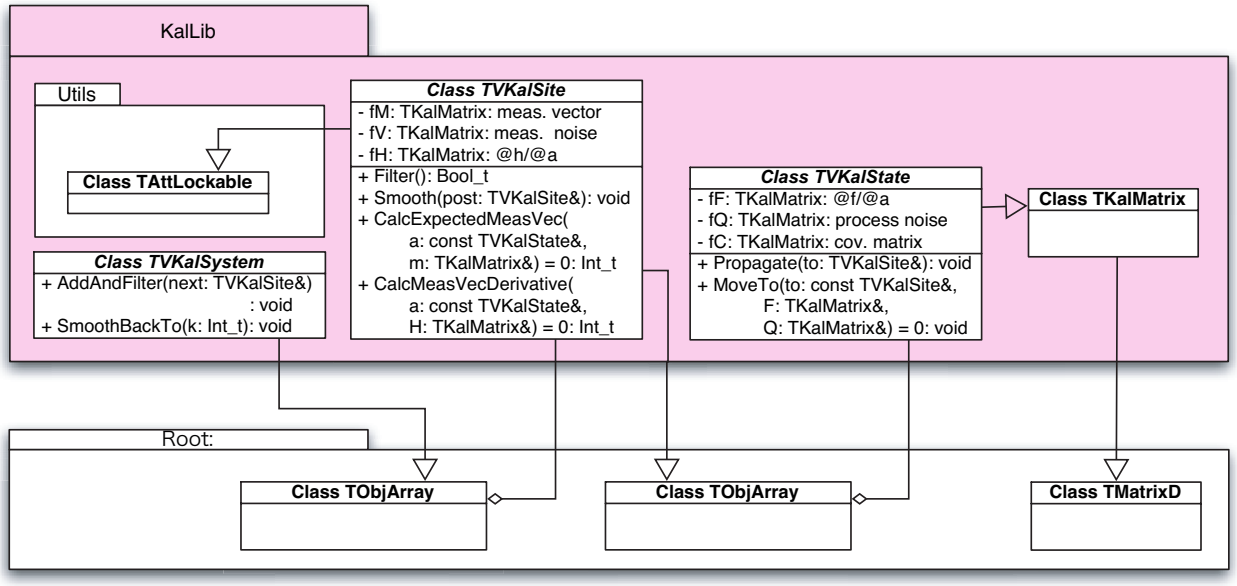

Figure 4.1: KalLib: Kalman filter base class library.

class representing a system to process with Kalman Filter. It is a ROOT's `TObjArray` of `TVKalSite`'s. A `TVKalSite` object corresponds to a measurement site and is a `TObjArray` of up to three `TVKalState`'s, corresponding to predicted, filtered, and smoothed state vectors. On the other hand, a `TVKalState` object stands for a state vector and is a $p \times 1$ `TKalMatrix`, which inherits from ROOT's `TMatrixD`. These abstract classes have to carry $\boldsymbol{f}_{k-1}(\boldsymbol{a}_{k-1})$, $\boldsymbol{F}_{k-1}$, $\boldsymbol{Q}_{k-1}$, $\boldsymbol{m}_k$, $\boldsymbol{H}_k$, and $\boldsymbol{V}_k$ as data members and corresponding pure virtual functions to evaluate them. In what follows, we will explain major data members and member functions of these abstract classes so as to demonstrate which abstract class carries which data members and member functions.

4.2.1 TVKalSystem Class (Inheriting from TObjArray; an Array of Sites)

The `TVKalSystem` class inherits from ROOT's `TObjArray` and hence behaves as an array of measurement sites. It is the abstract base class representing any Kalman system and is the only class that can handle relation between different sites.

Data Members:

```
TVKalSite * fCurSitePtr; // a pointer to the current site.
Double_t    fChi2;       // χ² value.
```

Member Functions:

```
void AddAndFilter(TVKalSite & next);
```

filters the next site (`next`) and add it to the base `TObjArray`.

```
void SmoothBackTo(Int_t k);
```

smoothes back from the last site to site (k).

4.2.2 TVKalSite Class (Inheriting from TObjArray: an Array of States: Measurement Site)

The `TVKalSite` class inherits from ROOT's `TObjArray` and hence behaves as an array that carries state vectors at various stages (Predicted, Filtered, and Smoothed). It also inherits from `TAttLockable` in the Utils package so that it can be excluded from filtering without being eliminated from `TVKalSystem`. It is the abstract base class representing a measurement site and stores, as data members, information on the observation made there and the estimates of the state vector thereat.

Data Members:

```
TKalMatrix fM; // measurement vector.
TKalMatrix fV; // measurement error matrix.
TKalMatrix fH; // projector matrix.
```

Member Functions:

```
Int_t CalcExpectedMeasVec(const TVKalState & a, TKalMatrix & h) = 0;
```

A pure virtual function representing a projector to calculate the expected measurement vector `h` for a given state vector `a`. It is application-specific and to be implemented in a derived class.

```
Int_t CalcMeasVecDerivative(const TVKalState & a, TKalMatrix & H) = 0;
```

A pure virtual function representing a projector matrix `H`, which is the derivative of the projector with respect to the input state vector `a`. It is application-specific and to be implemented in a derived class.

```
Bool_t Filter();
```

implements the generic algorithm of Kalman Filter as expressed by Eq.(3.1). It filters this instance of `TVKalSite` by using a predicted state vector from the previous site, a measurement vector, and its error matrix stored in the instance itself, calculates the filtered state vector $\boldsymbol{a}_k$, and its covariance matrix $\boldsymbol{C}_k$, and adds them to the base class `TObjArray` as a `TVKalState`.

```
void Smooth(TVKalSite & post);
```

implements the generic algorithm of Smoothing as given by Eqs.(2.34) and (2.35). It

smoothes this instance of `TVKalSite` by using the information stored in the instance itself and that from the post site (`post`), calculates the smoothed state vector $\boldsymbol{a}_k^n$ and its covariance matrix $\boldsymbol{C}_k^n$, and adds them to the base class `TObjArray` as a `TVKalState`.

4.2.3 TVKalState Class (Inheriting from TKalMatrix: State Vector)

The `TVKalState` class is a $p \times 1$ `TKalMatrix`, which in turn inherits from ROOT's `TMatrixD` (a double precision matrix). It is the abstract base class representing a state vector stored in the base class TKalMatrix.

Data Members:

```
TVKalSite  * fSitePtr;   // pointer to corresponding site.
TKalMatrix   fF;         // propagator matrix.
TKalMatrix   fQ;         // covariance matrix for the process noise.
TKalMatrix   fC;         // covariance matrix for the state vector.
```

Member Functions:

```
TVKalState & MoveTo(const TVKalSite & to, TKalMatrix & F,
                          TKalMatrix & Q) const = 0;
```

A pure virtual function that transports this instance of `TVKalState` to the destination site (`to`) and returns itself as reference, together with the corresponding propagator matrix and the process noise matrix. It is application-specific and to be implemented in a derived class.

```
void Propagate(TVKalSite & to);
```

propagates this instance of `TVKalState` to the destination site (`to`) and calculates the covariance matrix of the predicted state vector there. It implements the generic algorithm (Eqs.(2.9) and (2.13)) to propagate a state vector and its covariance matrix by means of the `MoveTo()` function.

4.3 Kaltracklib: Kalman-Filter-Based Track Fitting Library

In this section, we will apply the Kalman filter technique to track fitting by implementing, in a way specific to track fitting, pure virtual member functions of the base classes such as `TVKalSystem`, `TVKalSite`, and `TVKalState` in KalLib in their derived classes. These derived classes comprise, together with some other supplementary classes, a class library called KalTrackLib. Fig.4.2 is a class diagram showing the architecture of KalTrackLib. The classes defined in KalTrackLib implement the common algorithm specific to track fitting but are not dependent on any particular shape or coordinate system of tracking devices in question. In what follows, we will explain essential features , major data members and member functions relevant to end users, of the classes in KalTrackLib.

4.3.1 TKalTrack Class: Inheriting from TVKalSystem; Track

A track is a set of hit points. It is, hence, natural to define a track as an instance of a class inheriting from `TVKalSystem`, which is an array of measurement sites. Since `TVKalSystem` has no pure virtual functions, there is no base class member function to override.

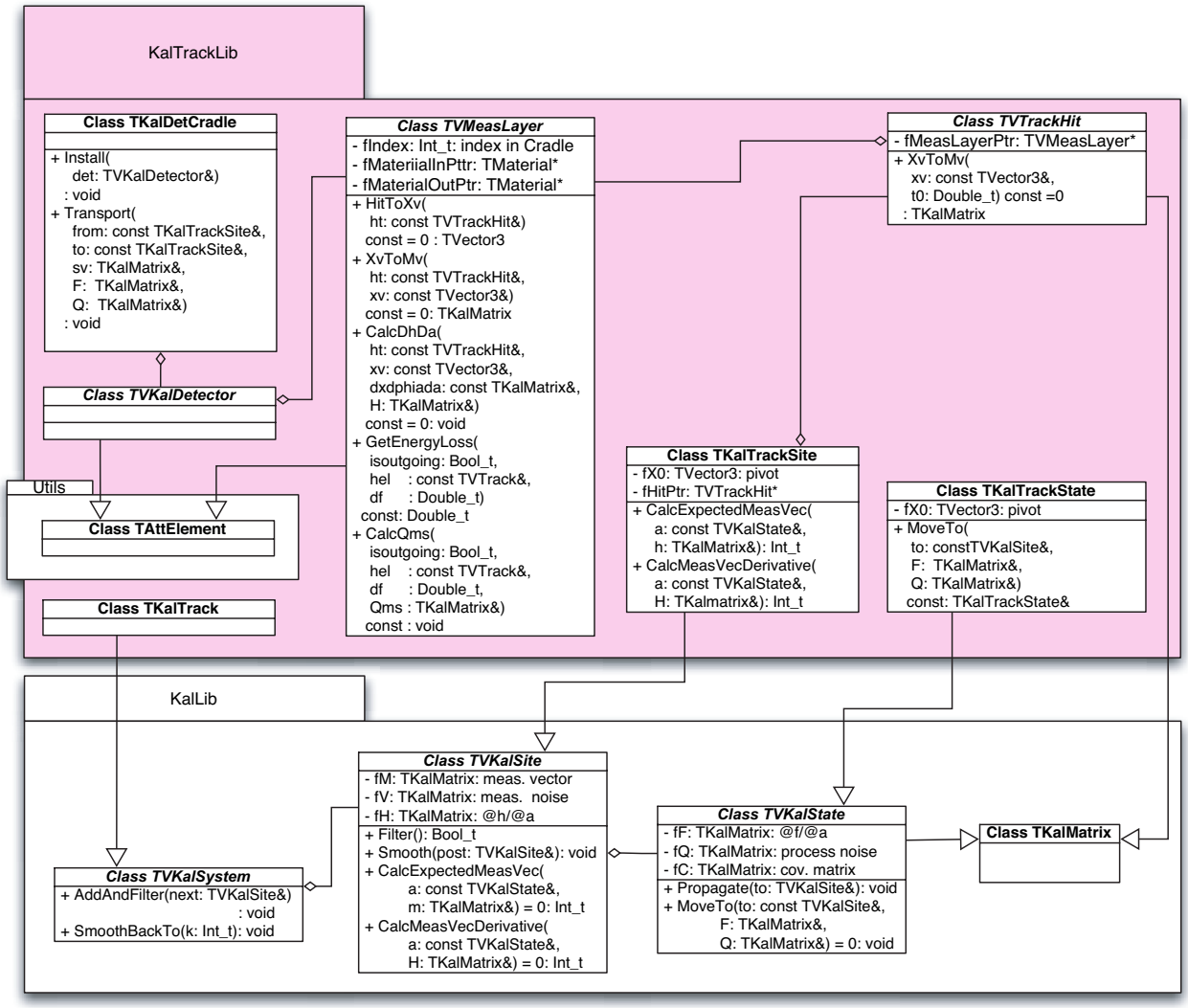

Figure 4.2: KalTrackLib: Kalman-filter-based track fitting class library.

Member Functions:

```
TKalTrack();
```

constructor that constructs an instance of `TKalTrack`.

4.3.2 TKalTrackSite Class: Inheriting from TVKalSite; Measurement Site

In track fitting, a measurement site provides a hit point, whose coordinates comprise a measurement vector. The `TKalTrackSite` class is designed to serve as the measurement site and implements the pure virtual member functions of the base class, `TVKalSite`, while carrying a pointer to a measurement vector (`TVTrackHit`) as one of its data members.

Data Members:

```
TVTrackHit * fHitPtr; // pointer to a corresponding hit object.
TVector3     fX0;     // pivot set to the hit point.
```

Member Functions:

```
TKalTrackSite(const TVTrackHit & ht);
```

A constructor that constructs an instance from a hit object.

```
Int_t CalcExpectedMeasVec(const TVKalState & a, TKalMatrix & h);
```

implements its base class pure virtual function that provides a projector to calculate the expected measurement vector `h` for a given state vector `a`. This function realizes the algorithm in subsection 3.3.1 and the Newtonian method in subsection 3.3.2.

```
Int_t CalcMeasVecDerivative(const TVKalState & a, TKalMatrix & H);
```

implements its base class pure virtual function that provides a projector matrix `H`, which is the derivative of the projector with respect to the input state vector `a`. This function realizes the algorithm in subsection 3.3.2.

Functions for derivatives such as $\partial \boldsymbol{x}/\partial \phi_k$ and $\partial \boldsymbol{x}/\partial \boldsymbol{a}$ that appear in the realization of the generic algorithm in `CalcExpectedMeasVec()` and `CalcMeasVecDerivative()` depend on a track model (trajectory equation). Their interface will be fixed by an abstract base class, `TVTrack`, defined later in GeomLib and be implemented in a derived class that describes a particular track model such as a helix or a straight line. The derivative of the measurement vector with respect to the position vector $\partial \boldsymbol{m}_k/\partial \boldsymbol{x}$ is, on the other hand, determined by the shape and coordinate system of the measurement layer in question. The interface of the function to calculate it will be specified by an abstract base class, `TVMeasLayer`, defined later and be implemented in a derived class that describes a particular measurement layer.

4.3.3 TKalTrackState Class: Inheriting from TVKalState; Track Parameter Vector

In track fitting, a state vector carries track parameters as their components. `TKalTrackState` is designed to serve as the track parameter vector and implements the pure virtual member function MoveTo() of the base class, `TVKalState`.

Data Members:

```
TVector3    fX0;     // pivot.
```

Member Functions:

```
TKalTrackState & MoveTo(const TVKalSite  & to,
                              TKalMatrix & F,
                              TKalMatrix & Q) const;
```

implements the base class pure virtual function that transports this instance of `TKalTrackState` to the destination site (`to`) and returns itself as reference, together with the corresponding propagator matrix and the process noise matrix.

The calculation of the process noise depends on the distribution of materials between the current site and the next site. As described below the structure of a tracking device is defined by a concrete class derived from an abstract base class, `TVKalDetector`, which is an array of measurement layers that derive from their abstract base class `TVMeasLayer`. The information on the local materials about the measurement layer is stored in the `TVMeasLayer` class. If necessary, we can compose a tracking system consisting of multiple tracking devices by installing the corresponding KalDetectors into a folder class, `TKalDetCradle`. The instance of `TKalDetCradle` then knows everything about the detector configuration. The `MoveTo()` function thus invokes `TKalDetCradle::Transport()` to move to site `to` in order to take into account the process noise that depends on the material distribution. Notice that the `TKalDetCradle::Transport()` functions invokes a pure virtual `MoveTo()` function of an abstract class, `TVTrack`, in GeomLib so that the track model can change from site to site in accordance with the local magnetic field.

4.3.4 TVTrackHit Class: Inheriting from TKalMatrix; Measurement Vector

In track fitting, a hit point plays the role of a measurement vector. It provides a set of coordinates at the intersection of a particle's trajectory with a measurement layer. The `TVTrackHit` class is designed to be the abstract base class for any hit point and specifies, through a pure virtual method (`XvToMv()`), the interface for the coordinate projector from the 3-dimensional position vector corresponding to the hit point to coordinate axes of the measurement layer.

Data Members:

```
Double_t      fBfield;       // magnetic field.
TVMeasLayer * fMeasLayerPtr; // pointer to the corresponding measurement layer.
```

Member Functions:

```
TKalMatrix XvToMv(const TVector3 & xv, Double_t t0) const = 0;
```

> A pure virtual function that specifies the interface for the coordinate projector from the expected hit position vector `xv` to the coordinate axes of the measurement layer, knowing the measured coordinates as stored in the instance and the input time stamp `t0` of the hit. The coordinate projector depends on the coordinate system of the measurement layer in question and hence is to be implemented in a derived class.

In order to introduce a new tracking device, one has to define a concrete hit point class inheriting from `TVTrackHit` and implements its pure virtual method, `XvToMv()`.

4.3.5 TVMeasLayer Class: Measurement Layer

As mentioned earlier, a tracking detector can be regarded as a set of measurement layers. Each of these measurement layers should be equipped with at least common interfaces for the projector $\boldsymbol{h}_k$ and the projector matrix $\boldsymbol{H}_k$ and for various other functions needed to calculate them using the generic algorithms explained in subsections 3.3.1 and 3.3.2. These common interfaces are specified by abstract base classes, `TVMeasLayer` in KalTrackLib and `TVSurface` in GeomLib. A concrete measurement layer then multiply inherits from `TVMeasLayer` and a concrete surface object such as a cylinder (`TCylinder`) or a hyperboloid (`THype`) or a plane (`TPlane`) derived from `TVSurface`, thereby realizing a particular shape and a coordinate system specific to the measurement layer. The `TVMeasLayer` class inherits from `TAttElement` and behaves as an element of some container, a `TVKalDetector` object described below.

Data Members:

```
TMaterial    * fMaterialInPtr;        // pointer to inner material.
TMaterial    * fMaterialOutPtr;       // pointer to out ner material.
Int_t          fIndex;                // index in TKalDetCradle.
Bool_t         fIsActive;             // flag to tell layer is active or not.
```

Notice that a measurement layer defines a surface on which coordinate measurements are made or, if it is not active, it just spcifies a boundary between different materials.

Member Functions:

```
TKalMatrix XvToMv(const TVTrackHit & ht, const TVector3 & xv) const = 0;
```

A pure virtual function corresponding to Eq.(3.18) that projects the 3-dimensional hit point vector `xv` to coordinate axes and returns the resultant measurement vector.

```
TVector3 HitToXv(const TVTrackHit & ht) const = 0;
```

A pure virtual function that transforms a measurement vector as stored in `ht` to a 3-dimensional hit point vector. To be used to calculate a next pivot.

```
void CalcDhDa(const TVTrackHit & ht,
              const TVector3   & xv,
              const TKalMatrix & dxphiada,
                    TKalMatrix & H) const = 0;
```

A pure virtual function that calculates the projector matrix `H` at the hit point `xv` with `dxphiada` $= \partial \boldsymbol{x}(\phi(\boldsymbol{a}), \boldsymbol{a})/\partial \boldsymbol{a}$ given as an input, assuming the use of the generic algorithm given in subsection 3.3.2.

```
Double_t CalcEnergyLoss(      Bool_t     isoutgoingt,
                        const TVTrack  & hel,
                              Double_t   df) const;
```

calculates and returns energy loss during the evolution of the track `hel` by the deflection angle `df`, knowing the direction of flight specified by `isoutgoing`.

```
void CalcQms(      Bool_t       isoutgoingt,
             const TVTrack    & hel,
                   Double_t     df,
                   TKalMatrix & Qms) const;
```

calculates the process noise matrix $\boldsymbol{Q}_m$ as given by Eq.(3.13) for the evolution of the track `hel` by the deflection angle `df`, knowing the direction of flight specified by `isoutgoing`.

As mentioned above, a concrete measurement layer multiply inherits from this `TVMeasLayer` class and a concrete surface object derived from `TVSurface`. Since the surface objects such as `TCylinder` or `THype` or `TPlane` are already equipped with methods to calculate the intersection of a track and a surface, etc., all users have to implement in the concrete measurement layer class is the projector to coordinate axes as given by Eq.(3.18) and the corresponding projector matrix.

4.3.6 TVKalDetector Class: Inheriting from TObjArray; Detector Component

In order to describe a tracking detector such as VTX, IT, and CT, consisting of multiple measurement layers implemented as instances of a concrete class that multiply inherits from the abstract base class `TVMeasLayer` and a concrete surface class such as `TCylinder` or `THype` or `TPlane`, we define an abstract base class called `TVKalDetector`, which inherits from ROOT's `TObjArray` and carries these concrete measurement layers as array elements. The `TVKalDetector` class acts only as a container and has no special member functions. The `TVKalDetector` class also inherits from `TAttElement` and behaves as an element of an instance of `TKalDetCradle` described below.

Notice that one can `Add()` a measurement layer with any shape or any coordinate system to `TVKalDetector`, as far as the measurement layer has `TVMeasLayer` and `TVSurface` as its base classes. This way one can compose a detector system with measurement layers with various shapes and coordinate systems in a unified manner.

4.3.7 TKalDetCradle Class: Inheriting from TObjArray; Detector System

In order to describe a whole tracking system consisting of tracking detector components such as VTX, IT, and CT, we define a class called `TKalDetCradle`, which inherits from ROOT's `TObjArray` and carries the individual measurement layers from various detector components as its array elements. The whole purpose of `TKalDetCradle` is to hold all the measurement layers including dummy layers corresponding to a beam pipe, support cylinders, etc. The `TKalDetCradle` object is the only object that knows everything about the distribution of materils in the detector system and hence its `Transport()` member function is used by `TKalTrackState::MoveTo()`.

Member Functions:

```
TKalDetCradle();
```

A constructor that constructs an instance of `TKalDetCradle`.

```
void Install(TVKalDetector & det);
```

installs `det` as a component of this instance of `TKalDetCradle`.

```
void Transport(const TKalTrackSite & from,
               const TKalTrackSite & to,
                     TKalMatrix    & sv,
                     TKalMatrix    & F,
                     TKalMatrix    & Q) const;
```

transports the input state `sv` from site `from` to site `to` and calculates the propagator matrix `F` and the process noise `Q` with successive thin layer multiple scattering approximation given by Eqs.(3.13) and (3.16).

4.4 GeomLib: Geometry Class Library

GeomLib is a library consisting of classes to describe track models and measurement surfaces. A class to represent a concrete track model such as a helix or a straight line should inherit from an abstract base class called `TVTrack`, while a class to represent a concrete surface such as a cylinder or a plane should inherit

from an abstract base class called `TVSurface`. These abstract base classes serve to set interfaces. Fig.4.3 is a class diagram showing the architecture of GeomLib. The classes defined in GeomLib implement utility

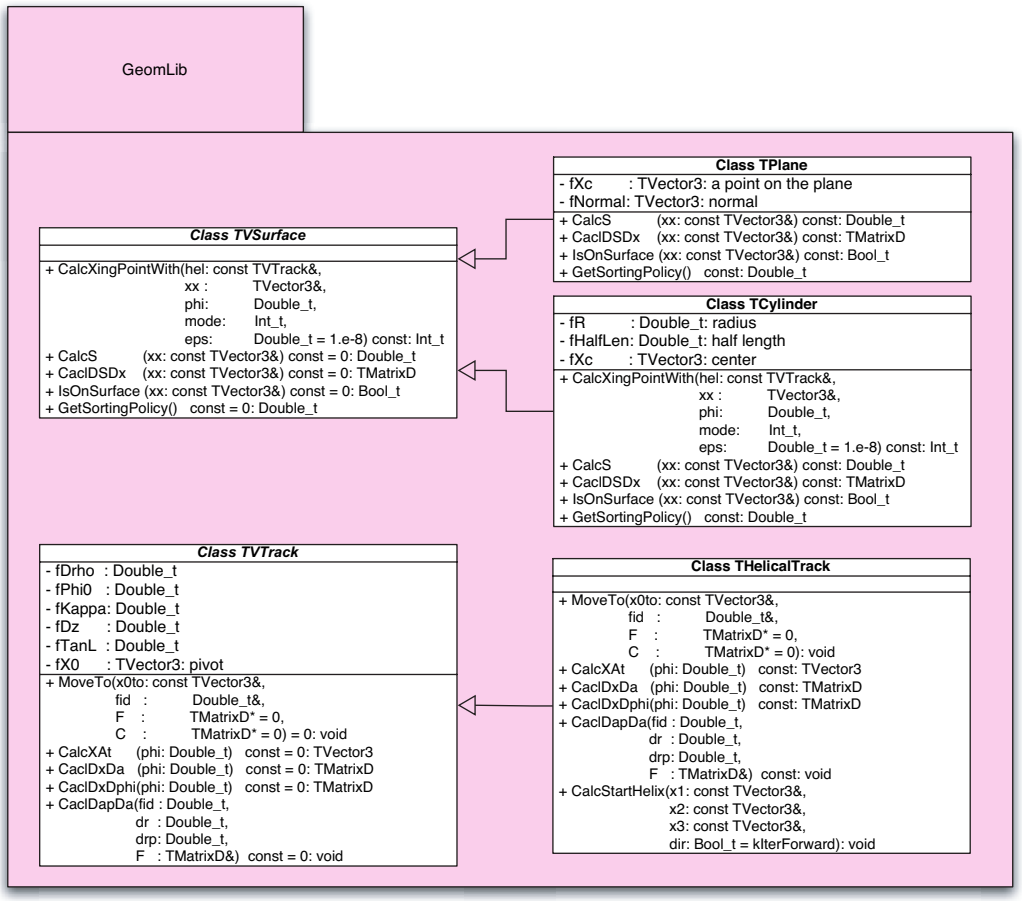

Figure 4.3: GeomLib: geometry class library to describe track models and measurement surfaces.

methods to calculate the intersection of a track and a surface, the tangential vector of a track, the gradient of a surface, etc., which are independent of KalTrackLib or KalLib. In what follows, we will explain essential features , major data members and member functions relevant to end users, of the classes in GeomLib.

4.4.1 TVTrack Class: Inheriting from TVCurve; Track

The `TVTrack` class is the abstract base class that sets interface for a trajectory equation as a function of a track parameter $\boldsymbol{a}$ and a parameter ϕ that specifies location along the trajectory[1].

Data Members:

[1]The trajectory parameter ϕ is the deflection angle from the pivot in the case of a helical track. It is the signed path length from the pivot in the case of a straight line track.

```
Double_t    fDrho;        // d_ρ.
Double_t    fPhi0;        // φ_0.
Double_t    fKappa;       // κ.
Double_t    fDz;          // d_z.
Double_t    fTanL;        // tan λ.
Double_t    fAlpha;       // α ≡ 1/cB (see Eq.(3.3)).
```

Member Functions:

```
void MoveTo(const TVector3 & xv0,]
                  Double_t & fid,
                  TMatrixD * F = 0,
                  TMatrixD * C = 0) = 0;
```

A pure virtual function to move the pivot to `xv0` and calculate the parameter change `fid` to the new pivot together with `*F`$= \partial \boldsymbol{a}'/\partial \boldsymbol{a}$ and/or the covariance matrix at the new pivot `*C`, if `F`$\neq 0$ and/or `C`$\neq 0$.

```
TVector3 CalcXAt(Double_t phi) const = 0;
```

A pure virtual function to calculate the position vector of a particle along the track at `phi`.

```
TMatrixD CalcDxDa(Double_t phi) const = 0;
```

A pure virtual function to calculate and return $\partial \boldsymbol{x}(\phi, \boldsymbol{a})/\partial \boldsymbol{a}$ (see subsection 3.3.2).

```
TMatrixD CalcDxDphi(Double|_t phi) const = 0;
```

A pure virtual function to calculate and return the tangential vector to the trajectory: $\partial \boldsymbol{x}(\phi, \boldsymbol{a})/\partial \phi$ (see subsection 3.3.2).

4.4.2 TVSurface Class: Inheriting from TObject; Surface

The `TVSurface` class is the abstract base class for any surface and sets interfaces for the methods to calculate its intersection with a track, the gradient at a given point on it, etc., as needed in the generic algorithm explained in subsection 3.3.2.

Member Functions:

```
Int_t CalcXingPointWith(const TVTrack  & hel,
                              TVector3 & xx,
                              Double_t & phi,
                              Int_t      mode,
                              Double_t   eps = 1.e-8) const;
```

implements the Newtonian method in subsection 3.3.2 to calculate the intersection `xx` with a track `hel` and the corresponding location parameter `phi` with the tolerance `eps`. The `mode` parameter is dummy here. It returns the number of intersections.

```
Double|_t CalcS(const TVector3 & xx) const = 0;
```

A pure virtual function to calculate the left-hand side $S_k(\boldsymbol{x})$ of the surface equation (Eq.(3.22)) at $\boldsymbol{x}$ =xx.

```
TMatrixD CalcDSDx(const TVector3 & xx) const = 0;
```

A pure virtual function to calculate the gradient of $S_k(\boldsymbol{x})$: $\partial S_k/\partial \boldsymbol{x}$ at $\boldsymbol{x}$ =xx.

```
Bool_t IsOnSurface(const TVector3 & xx) const = 0;
```

A pure virtual function to test if xx is on the surface or not.

```
Double_t GetSortingPolicy() const = 0;
```

A pure virtual function to get the sorting policy with which to sort surfaces from inside to outside.

4.4.3 THelicalTrack Class: Inheriting from TVTrack; Helical Track

The THelicalTrack class inherits from TVTrack and implements its pure virtual functions, thereby realizing a helical track.

Member Functions:

```
void MoveTo(const TVector3 & xv0,
                  Double_t & fid,
                  TMatrixD * F = 0,
                  TMatrixD * C = 0);
```

moves the pivot to xv0 and calculates the deflection angle fid to the new pivot together with *F= $\partial \boldsymbol{a}'/\partial \boldsymbol{a}$ and/or the covariance matrix at the new pivot *C, if F$\neq$ 0 and/or C$\neq$ 0 as explained in section 3.2.

```
TVector3 CalcXAt(Double_t phi) const;
```

calculates the position vector of a particle along the helix at the deflection angle phi according to Eq.(3.2).

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

```
TMatrixD CalcDxDphi(Double|_t phi) const = 0;
```

calculates and returns the tangential vector to the trajectory:

$$\frac{\partial \boldsymbol{x}(\phi, \boldsymbol{a})}{\partial \boldsymbol{\phi}} = \begin{pmatrix} \frac{\partial x}{\partial \phi} \\ \frac{\partial y}{\partial \phi} \\ \frac{\partial z}{\partial \phi} \end{pmatrix} = -\left(\frac{\alpha}{\kappa}\right) \cdot \begin{pmatrix} -\sin(\phi_0 + \phi) \\ \cos(\phi_0 + \phi) \\ \tan\lambda \end{pmatrix} . \tag{4.1}$$

See subsection 3.3.2.

```
TMatrixD CalcDxDa(Double_t phi) const;
```

calculates and returns the track-parameter derivative of the trajectory:

$$\frac{\partial \boldsymbol{x}(\phi, \boldsymbol{a})}{\partial \boldsymbol{a}} = \begin{pmatrix} \cos\phi_0 & \sin\phi_0 & 0 \\ -\left(d_\rho + \frac{\alpha}{\kappa}\right)\sin\phi_0 + \frac{\alpha}{\kappa}\sin(\phi_0 + \phi) & \left(d_\rho + \frac{\alpha}{\kappa}\right)\cos\phi_0 - \frac{\alpha}{\kappa}\cos(\phi_0 + \phi) & 0 \\ -\frac{\alpha}{\kappa^2}(\cos\phi_0 - \cos(\phi_0 + \phi)) & -\frac{\alpha}{\kappa^2}(\sin\phi_0 - \sin(\phi_0 + \phi)) & \frac{\alpha}{\kappa^2}\phi\tan\lambda \\ 0 & 0 & 1 \\ 0 & 0 & -\frac{\alpha}{\kappa}\phi \end{pmatrix}^T .$$

See subsection 3.3.2.

```
void CalcDapDa(Double_t   fid,
               Double_t   dr,
               Double_t   drp,
               TMatrixD & F) const;
```

calculates `F` $= \partial \boldsymbol{a}'/\partial \boldsymbol{a}$ when the pivot is moved by $\Delta\phi_0 =$ `fid` and d_ρ changed from `dr` to `drp`. See subsection 3.2.2.

```
void CalcStartHelix(const TVector3 & x1,
                    const TVector3 & x2,
                    const TVector3 & x3,
                          Bool_t     dir = kIterForward) const;
```

finds a helix passing through 3 points `x1`, `x2`, and `x3` and sets the resultant helix parameters to itself, knowing the direction of flight specified by `dir`.

4.4.4 TStraightTrack Class: Inheriting from TVTrack; Straight Line Track

A straight line track class `TStraightTrack` can be defined as the $\alpha \to \infty$ and $\phi \to 0$ limit of the `THelicalTrack` class with

$$\frac{\alpha}{\kappa}\phi \quad = \quad \text{constant.}$$

If we define this constant to be $-\psi$, we have

$$\lim_{\phi\to 0}\begin{pmatrix}\frac{\alpha}{\kappa}(\cos\phi_0-\cos(\phi_0+\phi))\\ \frac{\alpha}{\kappa}(\sin\phi_0-\sin(\phi_0+\phi))\\ -\frac{\alpha}{\kappa}\tan\lambda\phi\end{pmatrix} = -\lim_{\phi\to 0}\frac{\alpha}{\kappa}\phi\begin{pmatrix}\frac{\cos(\phi_0+\phi)-\cos\phi_0}{\phi}\\ \frac{\sin(\phi_0+\phi)-\sin\phi_0}{\phi}\\ \tan\lambda\end{pmatrix} = \psi\begin{pmatrix}-\sin\phi_0\\ \cos\phi_0\\ \tan\lambda\end{pmatrix},$$

which leads us to the following equation of trajectory:

$$\boldsymbol{x}\ (\phi,\boldsymbol{a}) = \begin{pmatrix}x\\ y\\ z\end{pmatrix} = \begin{pmatrix}x_0 & + & d_\rho\cos\phi_0 & - & \phi\sin\phi_0\\ y_0 & + & d_\rho\sin\phi_0 & + & \phi\cos\phi_0\\ z_0 & + & d_z & + & \phi\tan\lambda\end{pmatrix} \tag{4.2}$$

with ϕ redefined to be ψ. Notice that ϕ now stands for the signed projected path length to the x-y plane measured from the pivot.

Member Functions:

```
void MoveTo(const TVector3 & xv0,
                  Double_t & fid,
                  TMatrixD * F = 0,
                  TMatrixD * C = 0);
```

moves the pivot to `xv0` and calculates the projected path length `fid` to the new pivot together with `*F`= $\partial\boldsymbol{a}'/\partial\boldsymbol{a}$ and/or the covariance matrix at the new pivot `*C`, if `F`$\neq 0$ and/or `C`$\neq 0$.

```
TVector3 CalcXAt(Double_t phi) const;
```

calculates the position vector of a particle along the track at the projected path length `phi` according to Eq.(4.2).

```
TMatrixD CalcDxDphi(Double|_t phi) const = 0;
```

calculates and returns the tangential vector to the trajectory:

$$\frac{\partial\boldsymbol{x}(\phi,\boldsymbol{a})}{\partial\boldsymbol{\phi}} = \begin{pmatrix}-\sin\phi_0\\ \cos\phi_0\\ \tan\lambda\end{pmatrix}. \tag{4.3}$$

```
TMatrixD CalcDxDa(Double_t phi) const;
```

calculates and returns the track-parameter derivative of the trajectory[2]:

$$\frac{\partial \boldsymbol{x}(\phi, \boldsymbol{a})}{\partial \boldsymbol{a}} = \begin{pmatrix} \cos\phi_0 & \sin\phi_0 & 0 \\ -d_\rho \sin\phi_0 - \phi\cos\phi_0 & d_\rho\cos\phi_0 - \phi\sin\phi_0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & \phi \end{pmatrix}^T .$$

```
void CalcDapDa(Double_t   fid,
               Double_t   dr,
               Double_t   drp,
               TMatrixD & F) const;
```

calculates `F` $= \partial \boldsymbol{a}'/\partial \boldsymbol{a}$ when the pivot is moved by $\Delta\phi_0 =$ `fid` and d_ρ changed from `dr` to `drp`.

4.4.5 TCylinder Class: Inheriting from TVSurface; Cylindrical Surface

The `TCylinder` class inherits from `TVSurface` and implements its pure virtual functions, thereby realizing a cylindrical surface.

Data Members:

```
Double_t    fR;              // radius.
Double_t    fHalfLen;        // half length.
TVector3    fXc;             // center: x_c = (x_c, y_c, z_c)^T.
```

Member Functions:

```
Int_t CalcXingPointWith(const TVTrack  & hel,
                              TVector3 & xx,
                              Double_t & phi,
                              Int_t      mode,
                              Double_t   eps = 1.e-8) const;
```

overrides, for CPU time economy, the base class's Newtonian method to calculate the intersection `xx` with a track `hel` and the corresponding location parameter `phi` with the tolerance `eps`, since an analytic solution is available. The (forward, closest, backward) crossing pint is chosen for `mode = (-1,0,+1)`, respectively. It returns the number of intersections.

```
Double|_t CalcS(const TVector3 & xx) const;
```

calculates the left-hand side $S_k(\boldsymbol{x})$ of the surface equation (Eq.(3.22)) at $\boldsymbol{x} =$`xx`:

$$S_k(\boldsymbol{x}) = (x - x_c)^2 + (y - y_c)^2 - \texttt{fR}^2 \tag{4.4}$$

[2]We continue to use the same parameter vector with five components $\boldsymbol{a} = (d_\rho, \phi_0, \kappa, d_z, \tan\lambda)^T$ as with a helical track, though the trajectory does not depend on κ, in order to facilitate the continuation from or to a helix.

```
TMatrixD CalcDSDx(const TVector3 & xx) const;
```

calculates the gradient of $S_k(\boldsymbol{x})$: $\partial S_k/\partial \boldsymbol{x}$ at $\boldsymbol{x}$ =xx:

$$\frac{\partial S_k}{\partial \boldsymbol{x}} = \left(\frac{\partial S_k}{\partial x} \quad \frac{\partial S_k}{\partial y} \quad \frac{\partial S_k}{\partial z} \right) = (\, 2(x - x_c) \quad 2(y - y_c) \quad 0 \,) \, . \tag{4.5}$$

```
Bool_t IsOnSurface(const TVector3 & xx) const;
```

tests if xx is within the length of the cylinder.

```
Double_t GetSortingPolicy() const;
```

returns the radius fR as the sorting policy with which to sort surfaces from inside to outside.

4.4.6 THype Class: Inheriting from TVSurface; Hyperboloidal Surface

The THype class inherits from TVSurface and implements its pure virtual functions, thereby realizing a single-leaf hyperboloidal surface.

Data Members:

```
Double_t    fR0;          // waist radius at z = 0.
Double_t    fHalfLen;     // half length.
TVector3    fXc;          // center: x_c = (x_c, y_c, z_c)^T.
Double_t    fTanA;        // tan A with A being the stereo angle of wires.
```

Member Functions:

```
Double|_t CalcS(const TVector3 & xx) const;
```

calculates the left-hand side $S_k(\boldsymbol{x})$ of the surface equation (Eq.(3.22)) at $\boldsymbol{x}$ =xx:

$$S_k(\boldsymbol{x}) = (x - x_c)^2 + (y - y_c)^2 - \texttt{fR}^2 - (z - z_c)^2 \, \texttt{fTanA}^2 \tag{4.6}$$

```
TMatrixD CalcDSDx(const TVector3 & xx) const;
```

calculates the gradient of $S_k(\boldsymbol{x})$: $\partial S_k/\partial \boldsymbol{x}$ at $\boldsymbol{x}$ =xx:

$$\frac{\partial S_k}{\partial \boldsymbol{x}} = \left(\frac{\partial S_k}{\partial x} \quad \frac{\partial S_k}{\partial y} \quad \frac{\partial S_k}{\partial z} \right) = (\, 2(x - x_c) \quad 2(y - y_c) \quad -2(z - z_c) \, \texttt{fTanA}^2 \,) \, . \tag{4.7}$$

```
Bool_t IsOnSurface(const TVector3 & xx) const;
```

tests if xx is on the surface and within the length of the hyperboloid.

```
Double_t GetSortingPolicy() const;
```

returns the radius fR0 as the sorting policy with which to sort surfaces from inside to outside.

4.4.7 TPlane Class: Inheriting from TVSurface; Flat Surface

The `TPlane` class inherits from `TVSurface` and implements its pure virtual functions, thereby realizing a flat surface.

Data Members:

```
TVector3    fXc;          // reference point on the plane: x_c = (x_c, y_c, z_c)^T.
TVector3    fNormal;      // outward normal: n = (n_x, n_y, n_z)^T.
```

Member Functions:

```
Double|_t CalcS(const TVector3 & xx) const;
```

calculates the left-hand side $S_k(\boldsymbol{x})$ of the surface equation (Eq.(3.22)) at $\boldsymbol{x}$ =`xx`:

$$S_k(\boldsymbol{x}) = (\boldsymbol{x} - \boldsymbol{x}_c) \cdot \boldsymbol{n} = (x - x_c)\, n_x + (y - y_c)\, n_y + (z - z_c)\, n_z \tag{4.8}$$

```
TMatrixD CalcDSDx(const TVector3 & xx) const;
```

calculates the gradient of $S_k(\boldsymbol{x})$: $\partial S_k/\partial \boldsymbol{x}$ at $\boldsymbol{x}$ =`xx`:

$$\frac{\partial S_k}{\partial \boldsymbol{x}} = \boldsymbol{n} = (\, n_x \quad n_y \quad n_z \,)\,. \tag{4.9}$$

```
Bool_t IsOnSurface(const TVector3 & xx) const;
```

tests if `xx` is on the surface.

```
Double_t GetSortingPolicy() const;
```

returns the distance from the origin as the sorting policy with which to sort surfaces from inside to outside.

Chapter 5

How To Use the Kalman Filter Package

In this chapter we will show how to use the C++ class libraries elaborated in the last chapter, KalLib, KalTrackLib, and GeomLib, to apply the Kalman filter technique to track fitting with a particular tracking system, taking a simple example of a cylindrical tracking system distributed as `examples/kaltest/ct` in the KalTest package. As shown in Fig.5.1, the example includes the following three user-defined classes

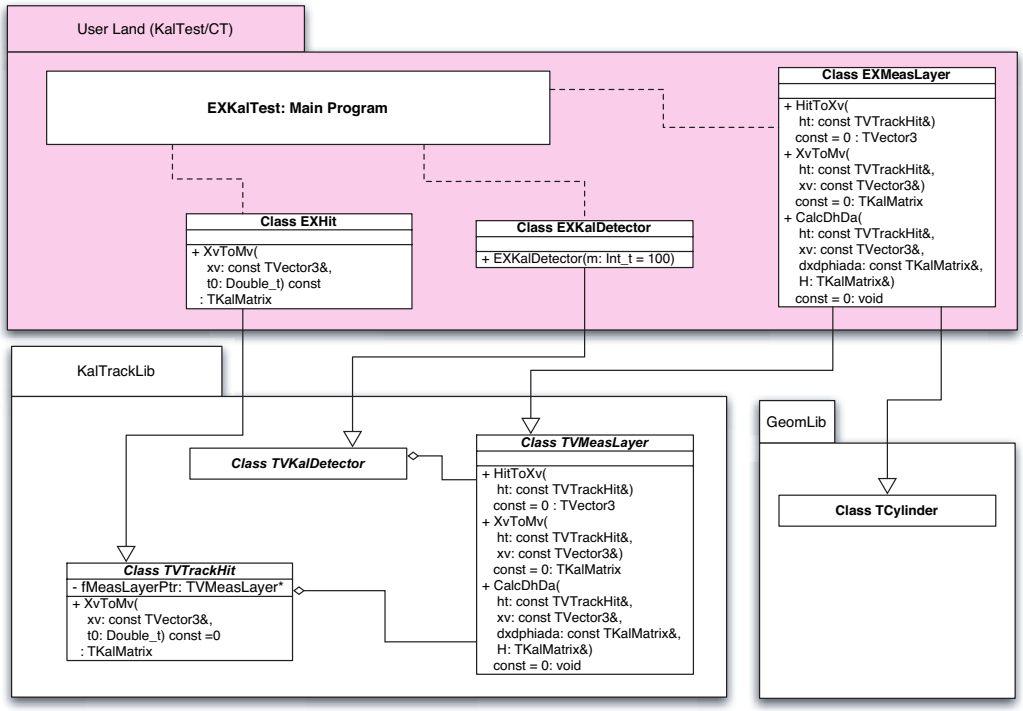

Figure 5.1: Sample code for a simple cylindrical tracker: `examples/kaltest/ct`.

required by the package:

- `EXMeasLayer`: a measurement layer that multiply inherits from the abstract measurement layer class `TVMeasLayer` and a concrete surface class `TCylinder` in GeomLib,
- `EXKalDetector`: an array that holds the user-defined measurement layers with particular shape and coordinate system and defines material distributions in the tracker, and

- `EXHit`: a measurement vector as defined by `EXMeasLayer`,

as well as a steering program to actually carry out the Kalman-filter-based track fitting `EXKalTest`. In what follows, we will explain essential features of these program units[1].

5.1 EXMeasLayer Class: Inheriting from TVMeasLayer and TCylinder

Let us start with a class `EXMeasLayer` to describe a simple cylindrical measurement layer having a radius R with its axis aligned with the z-axis. As explained in subsection 4.3.5, this suggests that `EXMeasLayer` is to inherits from `TVMeasLayer` in KalTrackLib and `TCyliner` in GeomLib.

For simplicity, let us assume that `EXMeasLayer` provides $R\phi$ and z coordinates of a hit point at $r = R$ with fixed measurement errors $\sigma_{r\phi}$ and σ_z, respectively. This sets the coordinate system necessary to implement the pure virtual functions of `TVMeasLayer` to calculate the projector to coordinate axes as given by Eq.(3.18) and the corresponding projector matrix.

5.1.1 Declaration of `EXMeasLayer` Class: `EXMeasLayer.h`

The header file for `EXMeasLayer` is shown below:

```
#ifndef __EXMEASLAYER__
#define __EXMEASLAYER__

#include "TVector3.h"     // from ROOT
#include "TKalMatrix.h"   // from KalLib
#include "TCylinder.h"    // from GeomLib
#include "TVMeasLayer.h"  // from KalTrackLib
#include "KalTrackDim.h"  // from KalTrackLib

class TVTrackHit; // forward declaration

class EXMeasLayer : public TVMeasLayer, public TCylinder {
public:
   // Ctors and Dtor
   EXMeasLayer(TMaterial &min,
               TMaterial &mout,
               Double_t   r0,
               Double_t   lhalf,
               Bool_t     isactive = kTRUE);
   virtual ~EXMeasLayer();

   // Parrent's pure virtuals that must be implemented
   virtual TKalMatrix XvToMv    (const TVTrackHit &ht,
                                 const TVector3   &xv) const;
   virtual TKalMatrix XvToMv    (const TVector3   &xv) const;
   virtual TVector3   HitToXv   (const TVTrackHit &ht) const;
```

[1] Notice that the sample code shown below might be slightly different from the actual code distributed with KalTest because of possible omissions of some features or details irrelevant for the purpose of this chapter.

```
   virtual void       CalcDhDa  (const TVTrackHit &ht,
                                 const TVector3   &xv,
                                 const TKalMatrix &dxphiada,
                                       TKalMatrix &H)  const;

   ClassDef(EXMeasLayer,1)  // Sample measurement layer class
};
#endif
```

where `ClassDef(...)` is a ROOT macro that expands into declarations for ROOT-required data fields and methods for automatic schema evolution.

5.1.2 Implementation of `EXMeasLayer` Class: Preamble

The implementation of `EXMeasLayer` starts with

```
#include "EXMeasLayer.h"
#include "EXHit.h"

ClassImp(EXMeasLayer)
```

where `ClassImp(EXMeasLayer)` is a ROOT macro that expands into implementations of ROOT-methods declared in the `ClassDef(...)` macro in the header file for automatic schema evolution.

5.1.3 Constructor and Destructor: `EXMeasLayer()` and ∼`EXMeasLayer()`

Then we need a constructor and the destructor:

```
EXMeasLayer::EXMeasLayer(TMaterial &min,      // material inside the layer
                         TMaterial &mout,     // material outside the layer
                         Double_t   r0,       // radius of the layer
                         Double_t   lhalf,    // half length of the layer
                         Bool_t     isactive) // flag to tell the layer is active
           : TVMeasLayer(min, mout, isactive),
             TCylinder(r0, lhalf)
{
}

EXMeasLayer::~EXMeasLayer()
{
}
```

where TMaterial is a ROOT class to store material information.

5.1.4 Coordinate Projector: `XvToMv()`

Then we need to implement the coordinate projector, which projects the hit point vector $\boldsymbol{x}_k$ to the coordinate axes to get the corresponding measurement vector:

$$\boldsymbol{m}_k(\boldsymbol{x}_k) \;=\; \begin{pmatrix} R_k \cdot \phi_k \\ z_k \end{pmatrix} = \begin{pmatrix} R_k \cdot \tan^{-1}\left(\frac{y_k}{x_k}\right) \\ z_k \end{pmatrix},$$

where R_k is the radius of layer (k). The coordinate projector method, `XvToMv(.....)`, calculates and returns the measurement vector `mv` as the return value:

```
TKalMatrix EXMeasLayer::XvToMv(const TVector3 &xv) const
{
   // Calculate hit coordinate information:
   // mv(0,0) = r * phi
   //     (1,0) = z

   TKalMatrix mv(kMdim,1);
   mv(0,0)  = GetR() * TMath::ATan2(xv.Y(), xv.X());
   mv(1,0)  = xv.Z();
   return mv;
}

TKalMatrix EXMeasLayer::XvToMv(const TVTrackHit & /* ht */,
                               const TVector3   &xv) const
{
   return XvToMv(xv);
}
```

where `GetR()` is the method of the base class `TCylinder`. Notice that in this example, `ht`, is not used.

5.1.5 Reverse Coordinate Projector: `HitToMv()`

We also need to implement the reverse coordinate projector, which projects back the measurement vector stored in an `EXHit` object to the 3-dimensional hit position vector:

$$\boldsymbol{x}_k \;=\; \begin{pmatrix} x_k \\ y_k \\ z_k \end{pmatrix} = \begin{pmatrix} R_k \cdot \cos\phi_k \\ R_k \cdot \sin\phi_k \\ z_k \end{pmatrix},$$

which can be realized for instance as follows:

```
TVector3 EXMeasLayer::HitToXv(const TVTrackHit &vht) const
{
   const EXHit &ht = dynamic_cast<const EXHit &>(vht);

   Double_t phi = ht(0,0) / GetR();
```

```
   Double_t z   = ht(1,0);
   Double_t x   = GetR() * TMath::Cos(phi);
   Double_t y   = GetR() * TMath::Sin(phi);

   return TVector3(x,y,z);
}
```

5.1.6 Projector Matrix: `CalcDhDa()`

Finally we need to implement the projector matrix:

$$\boldsymbol{H}_k \;=\; \begin{pmatrix} \frac{\partial(R_k\cdot\phi_k)}{\partial \boldsymbol{a}} \\ \\ \frac{\partial z_k}{\partial \boldsymbol{a}} \end{pmatrix} = \begin{pmatrix} R_k\left(\frac{\partial \phi_k}{\partial \boldsymbol{x}_k}\right)\left(\frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{a}}\right) \\ \\ \frac{\partial z_k}{\partial \boldsymbol{a}} \end{pmatrix} = \begin{pmatrix} -\frac{y_k}{R_k}\left(\frac{\partial x_k}{\partial \boldsymbol{a}}\right) + \frac{x_k}{R_k}\left(\frac{\partial y_k}{\partial \boldsymbol{a}}\right) \\ \\ \frac{\partial z_k}{\partial \boldsymbol{a}} \end{pmatrix},$$

where

$$\begin{aligned} \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{a}} \;=\; \frac{\partial \boldsymbol{x}\left(\phi\left(\boldsymbol{a}\right),\boldsymbol{a}\right)}{\partial \boldsymbol{a}} \;&=\; \frac{\partial \boldsymbol{x}}{\partial \phi_k}\cdot\frac{\partial \phi_k}{\partial \boldsymbol{a}} + \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{a}} \\ \frac{\partial \phi_k}{\partial \boldsymbol{a}} \;&=\; -\frac{1}{\left(\frac{\partial S_k}{\partial \boldsymbol{x}}\cdot\frac{\partial \boldsymbol{x}}{\partial \phi_k}\right)}\,\frac{\partial S_k}{\partial \boldsymbol{x}}\cdot\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{a}} \end{aligned}$$

as explained in subsection 3.3.2. The projector matrix method, `CalcDhDa(.....)`, can then be implemented for instance as folows:

```
void EXMeasLayer::CalcDhDa(const TVTrackHit &,          // Hit: not used here
                           const TVector3   &xxv,       // hit position vector
                           const TKalMatrix &dxphiada,  // @x(\phi(a),a)/@a
                                 TKalMatrix &H)  const  // projector matrix = @h/@a
{
   // Calculate
   //    H = (@h/@a) = (@phi/@a, @z/@a)^t
   // where
   //        h(a) = (phi, z)^t: expected measurement vector
   //        a = (drho, phi0, kappa, dz, tanl, t0)
   //

   Int_t sdim = H.GetNcols();
   Int_t hdim = TMath::Max(5,sdim-1);

   Double_t xv = xxv.X();
   Double_t yv = xxv.Y();
   Double_t xxyy = xv * xv + yv * yv;

   // Set H = (@h/@a) = (@d/@a, @z/@a)^t
```

```
      for (Int_t i=0; i<hdim; i++) {
         H(0,i) = - (yv / xxyy) * dxphiada(0,i)
                  + (xv / xxyy) * dxphiada(1,i);
         H(0,i) *= GetR();
         H(1,i) =  dxphiada(2,i);
      }
      if (sdim == 6) {
         H(0,sdim-1) = 0.;
         H(1,sdim-1) = 0.;
      }
   } ,
```

where in this example the measurement vector does not depend on `t0` and hence the 6th track parameter is dummy.

5.2 EXKalDetector Class: Inheriting from TVKalDetector

Now that we have the `EXMeasLayer` class implemented, we need a folder to construct and bind together the `EXMeasLayer` objects to build a tracking detector. For this purpose, let us implement the `EXKalDetector` class inheriting from `TVKalDetector` in KalTrackLib (see subsection 4.3.6).

5.2.1 Declaration of `EXKalDetector` Class: `EXKalDetector.h`

The header file for `EXMeasLayer` will then look like what follows:

```
#ifndef __EXKALDETECTOR__
#define __EXKALDETECTOR__

#include "TVector3.h"         // from ROOT
#include "TVKalDetector.h"    // from KalTrackLib
#include "EXMeasLayer.h"

class EXKalDetector : public TVKalDetector {
public:
   // Ctor and Dtor
   EXKalDetector();
   ~EXKalDetector();

   // Utility method
   Double_t   GetBfield (const TVector3 &xx = TVector3(0.)) const
                           { return fgBfield; }

private:
   static Double_t fgBfield;   // magnetic field [kG]
```

```
   ClassDef(EXKalDetector,1)   // Sample hit class
};

#endif
```

where `ClassDef(...)` is a ROOT macro that expands into declarations for ROOT-required data fields and methods for automatic schema evolution.

5.2.2 Implementation of `EXKalDetector` Class: Preamble

The implementation of `EXKalDetector` starts with

```
#include "EXKalDetector.h"
#include "EXMeasLayer.h"
#include "EXHit.h"

#include "TRandom.h"        // from ROOT

Double_t EXKalDetector::fgBfield = 30.;   // [kG]

ClassImp(EXKalDetector)
```

where `ClassImp(EXKalDetector)` is a ROOT macro that expands into implementations of ROOT-methods declared in the `ClassDef(...)` macro in the header file for automatic schema evolution.

5.2.3 Constructor and Destructor: `EXKalDetector()` and ∼`EXKalDetector()`

For simplicity, let us assume that our cylindrical tracker has a CFRP inner cylinder and multiple cylindrical measurement layers equally spaced in r surrounding the CFRP inner cylinder and that the inter-layer gaps are filled with air. We can construct such a structure in a constructor of `EXKalDetector`[2]:

```
EXKalDetector::EXKalDetector(Int_t m)
             : TVKalDetector(m)
{
   Double_t A, Z, density, radlen;
   A       = 14.00674 * 0.7 + 15.9994 * 0.3;    // mass number
   Z       = 7.3;                               // atomic number
   density = 1.205e-3;                          // [g/cmm^3]
   radlen  = 3.42e4;                            // [cm]
   TMaterial &air = *new TMaterial("Air", "", A, Z, density, radlen, 0.);
```

[2]ROOT forbids creation of any heap objects in the default constructor, since it will lead to memory leak in reading the class object from a flat file using ROOT's streamers. This example violates this rule since it does not perform any object I/O for the `EXKalDetector` class. If your `KalDetector` object needs to be written to or read from a flat file via ROOT's object I/O scheme, you need to move the creation of heap objects to somewhere outside the default constructor.

```
    A       = 12.0107;                          // mass number
    Z       =  6.;                              // atomic number
    density = 0.1317;                           // [g/cmm^3]
    radlen  = 42.7/density;                     // [cm]
    TMaterial &cfrp = *new TMaterial("CFRP", "", A, Z, density, radlen, 0.);

    static const Int_t    nlayers   = 50;       // # sampling layers
    static const Double_t lhalf     = 200.;     // half length
    static const Double_t rmin      = 45.;      // r_{min} = radius of 0th layer
    static const Double_t rstep     = 3.;       // step in r
    static const Double_t rcylin    = 43.;      // inner radius of CFRP cylinder
    static const Double_t rcylout   = 44.;      // outer radius of CFRP cylinder

    static const Bool_t kActive = kTRUE;
    static const Bool_t kDummy = kFALSE;

    // Create dummy layers of the inner cylinder of the central tracker
    Add(new EXMeasLayer(air, cfrp, rcylin, lhalf, kDummy));
    Add(new EXMeasLayer(cfrp, air, rcylout, lhalf, kDummy));

    // Create measurement layers of the central tracker
    Double_t r   = rmin;
    for (Int_t layer = 0; layer < nlayers; layer++) {
       Add(new EXMeasLayer(air, air, r, lhalf, kActive));
       r += rstep;
    }
    SetOwner();  // detector owns measurement layers
}

EXKalDetector::~EXKalDetector()
{
}
```

where TMaterial is a ROOT class to store material information.

5.3 EXHit Class: Inheriting from TVTrackHit

Having implemented the measurement layer class, `EXMeasLayer`, and its container or the detector consisting of it, `EXKalDetector`, we can now move on to a hit class on it, `EXHit`, derived from `TVTrackHit`.

5.3.1 Declaration of `EXHit` Class: `EXHit.h`

The header file for `EXHit` is shown below:

```
#ifndef __EXHIT__
#define __EXHIT__

#include "KalTrackDim.h"
#include "TVTrackHit.h"
#include "EXMeasLayer.h"

class EXHit : public TVTrackHit {
public:
   // Ctor and Dtor
   EXHit(const EXMeasLayer &ms,
               Double_t    *x,
               Double_t    *dx,
               Double_t     b,
               Int_t        m = kMdim);
   virtual ~EXHit();

   // Parent's pure virtual functions that must be implemented
   virtual TKalMatrix XvToMv (const TVector3 &xv, Double_t t0) const;
   virtual void       DebugPrint(Option_t *opt = "")           const;

   ClassDef(EXHit,1)      // Sample hit class
};
#endif
```

where `ClassDef(...)` is a ROOT macro that expands into declarations for ROOT-required data fields and methods for automatic schema evolution.

5.3.2 Implementation of `EXHit` Class: Preamble

The implementation of `EXHit` starts with

```
#include "EXHit.h"
#include "EXMeasLayer.h"

#include <iostream>
#include <iomanip>

using namespace std;

ClassImp(EXHit)
```

where `ClassImp(EXHit)` is a ROOT macro that expands into implementations of ROOT-methods declared in the `ClassDef(...)` macro in the header file for automatic schema evolution.

5.3.3 Constructor and Destructor: `EXHit()` and `~EXHit()`

Since the base class constructor requires, as inputs, a measurement layer, arrays of coordinate values and their errors, and a magnetic field value, we need a constructor that looks the following:

```
EXHit::EXHit(const EXMeasLayer &ms,    // measurement layer
                   Double_t    *x,     // coordinate array
                   Double_t    *dx,    // coordinate error array
                   Double_t     b,     // magnetic field
                   Int_t        m)     // dimension of meas. vector
     : TVTrackHit(ms, x, dx, b, m)
{
}

EXHit::~EXHit()
{
}
```

5.3.4 Coordinate Projector: `XvToMv()`

Then we need to implement the coordinate projector, which projects the input hit point vector `xv` to the coordinate axes to get the expected measurement vector:

```
TKalMatrix EXHit::XvToMv(const TVector3 &xv, Double_t /*t0*/) const
{
   const EXMeasLayer &ms
                = dynamic_cast<const EXMeasLayer &>(GetMeasLayer());
   TKalMatrix h  = ms.XvToMv(xv);
   Double_t r    = ms.GetR();
   Double_t phih = h(0,0) / r;
   Double_t phim = (*this)(0,0) / r;
   Double_t dphi = phih - phim;

   static Double_t kPi    = TMath::Pi();
   static Double_t kTwoPi = 2 * kPi;

   while (dphi < -kPi) dphi += kTwoPi;
   while (dphi >  kPi) dphi -= kTwoPi;

   h(0,0)  = r * (phim + dphi);
   h(1,0) += 0.;

   return h;
}
```

where `t0` is not used in this simple example. Notice that the function of this `XvToMv()` method is to avoid the wrap-around problem at the 2π boundary and to choose the correct ϕ that matches the measurement vector stored in the `EXHit` object.

5.3.5 Debugging Method: `DebugPrint()`

The `DebugPrint()` function is another pure virtual function in the base class, `TVTrackHit`, that must be implemented to facilitate debugging:

```
void EXHit::DebugPrint(Option_t *) const
{
   cerr << "------------------- Site Info -------------------------" << endl;

   for (Int_t i=0; i<GetDimension(); i++) {
      Double_t x  = (*this)(i,0);
      Double_t dx = (*this)(i,1);
      cerr << " x[" << i << "] = " << setw(8) << setprecision(5) << x
           << "    "
           << "dx[" << i << "] = " << setw(6) << setprecision(2) << dx
           << setprecision(7)
           << resetiosflags(ios::showpoint)
           << endl;
   }
   cerr << "-------------------------------------------------------" << endl;
}
```

where `GetDimension()` is a method of ROOT's `TMatrixD`.

5.4 EXKalTest: Main Program

Now that we have the measurement layer class, `EXMeasLayer`, the tracking detector class, `EXKalDetector`, and the hit class, `EXHit`, we can now perform Kalman-filter-based track fitting. The following is a sample main program to do it:

```
#include "TKalDetCradle.h"    // from KalTrackLib
#include "TKalTrackState.h"   // from KalTrackLib
#include "TKalTrackSite.h"    // from KalTrackLib
#include "TKalTrack.h"        // from KalTrackLib

#include "EXKalTest.h"
#include "EXKalDetector.h"
#include "EXEventGen.h"
#include "EXHit.h"

#include "TNtupleD.h"         // from ROOT
```

```
#include "TFile.h"            // from ROOT

#include <iostream>

static const Bool_t gkDir = kIterBackward;

int main (Int_t argc, Char_t **argv)
{
   // ====================================================================
   //  Initialize ROOT
   // ====================================================================

   gROOT->SetBatch();
   TApplication app("EXKalTest", &argc, argv, 0, 0);

   // ====================================================================
   //  Open a ROOT file and define an ntuple to store fit results
   // ====================================================================
   TFile hfile("h.root","RECREATE","KalTest");
   TNtupleD *hTrackMonitor = new TNtupleD("track", "", "ndf:chi2:cl:cpa");

   // ====================================================================
   //  Prepare a detector
   // ====================================================================

   TKalDetCradle cradle;     // detctor system
   EXKalDetector detector;   // CT detector

   cradle.Install(detector); // install detector into its cradle
#ifdef __MS_OFF__
   cradle.SwitchOffMS();     // switch off multiple scattering
#endif

   // ====================================================================
   //  Loop over events
   // ====================================================================

   Int_t    nevents = 100;
   for (Int_t eventno = 0; eventno < nevents; eventno++) {
      cerr << "------ Event " << eventno << " ------" << endl;

     TObjArray     kalhits;    // hit buffer

      // ------------------------------------------------------------
      //  Generate an event, create EXHit's on EXMeasLayer's, and
      //  store them in kalhits.
      // ------------------------------------------------------------
```

```
// ..........
//      code not shown
// ..........

// ============================================================
//  Do Kalman Filter
// ============================================================
// ---------------------------
//  Prepare hit iterrator
// ---------------------------
TIter next(&kalhits, gkDir);

// ---------------------------
//  Create a dummy site: sited
// ---------------------------

EXHit &hitd = *static_cast<EXHit *>(next());
hitd(0,1) = 1.e6;   // give a huge error to d
hitd(1,1) = 1.e6;   // give a huge error to z
next.Reset();       // rewind iterator

TKalTrackSite &sited = *new TKalTrackSite(hitd);
sited.SetOwner();   // site owns states

// ---------------------------
//  Create an initial helix
// ---------------------------

Int_t i1, i2, i3;
if (gkDir == kIterBackward) {
   i3 = 0;
   i1 = kalhits.GetEntries() - 1;
   i2 = i1 / 2;
} else {
   i1 = 0;
   i3 = kalhits.GetEntries() - 1;
   i2 = i3 / 2;
}
EXHit   &h1 = *dynamic_cast<EXHit *>(kalhits.At(i1));   // first hit
EXHit   &h2 = *dynamic_cast<EXHit *>(kalhits.At(i2));   // last hit
EXHit   &h3 = *dynamic_cast<EXHit *>(kalhits.At(i3));   // middle hit
TVector3 x1 = h1.GetMeasLayer().HitToXv(h1);
TVector3 x2 = h2.GetMeasLayer().HitToXv(h2);
TVector3 x3 = h3.GetMeasLayer().HitToXv(h3);
THelicalTrack helstart(x1, x2, x3, h1.GetBfield());   // initial helix
```

```
// ---------------------------
//  Set dummy state to sited
// ---------------------------

static TKalMatrix svd(kSdim,1);
svd(0,0) = 0.;
svd(1,0) = helstart.GetPhi0();
svd(2,0) = helstart.GetKappa();
svd(3,0) = 0.;
svd(4,0) = helstart.GetTanLambda();
if (kSdim == 6) svd(5,0) = 0.;

static TKalMatrix C(kSdim,kSdim);
for (Int_t i=0; i<kSdim; i++) {
   C(i,i) = 1.e4;   // dummy error matrix
}

sited.Add(new TKalTrackState(svd,C,sited,TVKalSite::kPredicted));
sited.Add(new TKalTrackState(svd,C,sited,TVKalSite::kFiltered));

// ---------------------------
//  Add sited to the kaltrack
// ---------------------------

TKalTrack kaltrack;    // a track is a kal system
kaltrack.SetOwner();   // kaltrack owns sites
kaltrack.Add(&sited);  // add the dummy site to the track

// ---------------------------
//  Start Kalman Filter
// ---------------------------

EXHit *hitp = 0;
while ((hitp = dynamic_cast<EXHit *>(next()))) {     // loop over hits
   TKalTrackSite  &site = *new TKalTrackSite(*hitp); // create a site
   if (!kaltrack.AddAndFilter(site)) {               // add and filter it
      cerr << " site discarded!" << endl;
      delete &site;                                  // delete it, if failed
   }
}
//kaltrack.SmoothBackTo(3);                          // smooth back.

// ============================================================
//  Monitor Fit Results (ndf, chi2, confidence level, kappa)
// ============================================================

Int_t    ndf  = kaltrack.GetNDF();
```

```
         Double_t chi2 = kaltrack.GetChi2();
         Double_t cl   = TMath::Prob(chi2, ndf);
         Double_t cpa  = kaltrack.GetState(TVKalSite::kFiltered)(2, 0);
         hTrackMonitor->Fill(ndf, chi2, cl, cpa);
      }

      // ============================================================
      //  Write the ntuple to the output ROOT file
      // ============================================================
      hfile.Write();

      return 0;
   }
```

where the event generation and the hit point making part of the program is omitted.

Bibliography

[1] R. E. Kalman, J. Basic Eng. 82 (1961) 34.

[2] R. Frühwirth, Nucl. Instr. and Meth. **A262** (1987) 444.

[3] E. J. Wolin and L. L. Ho, Nucl. Instr. and Meth. **A219** (1993) 493.

[4] P. Astier *et al.*, Nucl. Instr. and Meth. **A450** (2000) 138.

[5] Helix Manipulation, the JLC Group, internal circulation (1998).