2.2. Kalman Filter Track Fit

Exercises are located at Exercises/2_SIMD/
Solutions are located at Exercises/2_SIMD/Solutions/
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

2_SIMD/3_KF: description

The Kalman filter is a method of obtaining estimate of unknown variable that uses a series of noisy measurements observed over time. The resulting estimate tend to be more precise than estimates based on a single measurements alone. The filter is named after Rudolf Kalman, one of the primary developers of its theory.

The Kalman filter has numerous applications in technology and science. A common application is for guidance, navigation and control of vehicles, particularly aircrafts and spacecrafts. Furthermore, the Kalman filter is a widely applied concept used in fields such as signal processing and econometrics.

The Kalman filter is a recursive estimator – only the estimated state from the previous time step and the current measurement are needed to compute the estimate for the current state. For illustration let us reformulate a calculation of a mean value of N elements in a recursive form. The general form of mean value is defined as:

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The mean over n+1 elements can be rewritten in a way:

$$\mu_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{n}{n+1}\left(\frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n}x_{n+1}\right) = \frac{n}{n+1}\mu_n + \frac{1}{n+1}x_{n+1} = \mu_n + \frac{1}{n+1}(x_{n+1} - \mu_n)$$

Now the mean of n+1 is determined by the previous estimate and a correction term, which is a new measurement with a weighting coefficient 1/(n+1). After we have reformulated the problem in a recursive way Kalman filter method becomes applicable.

The Kalman filter method is intended for finding the optimum estimation $\mathbf{r}$ of an unknown state vector of a system $\mathbf{r}^t$ based on k measurements m_k, $k = 1,\dots,n$ by minimising the mean square estimation error. The estimation $\mathbf{r}$ is known with the error $\boldsymbol{\xi}$:

$\mathbf{r} = \mathbf{r}^t + \boldsymbol{\xi}$,

therefore the covariance matrix of the estimation is introduced:

$C = \langle \boldsymbol{\xi} \cdot \boldsymbol{\xi}^T \rangle$.

The state vector is normally not observed directly, but through the detector measurements. Let's assume that the measurement $\mathbf{m}_k$ linearly depends on $\mathbf{r}^t_k$:

$\mathbf{m}_k = H_k \mathbf{r}^t_k + \boldsymbol{\eta}_k$, where H_k is the measurement model and $\boldsymbol{\eta}_k$ is an error of the k-th measurement.

The evolution of the linear system proceeds in space from one measurement $\mathbf{m}_{k-1}$ to the next measurement $\mathbf{m}_k$ and is described by a linear equation:

$\mathbf{r}^t_k = F_{k-1}\mathbf{r}^t_{k-1} + \mathbf{v}_k$,

where F_{k-1} is a linear propagation operator, $\mathbf{v}_k$ is a random process noise between the measurements $\mathbf{m}_{k-1}$ and $\mathbf{m}_k$.

The measurement errors $\boldsymbol{\eta}_k$ and the process noise $\mathbf{v}_k$ are assumed to be uncorrelated and unbiased, and those covariance matrices V_k and Q_k are known:

$\langle \boldsymbol{\eta} \rangle = \langle \mathbf{v} \rangle = 0$,

$\langle \boldsymbol{\eta}_k \cdot \boldsymbol{\eta}^T{}_k \rangle \equiv V_k$, (4b) $\langle \mathrm{v}_k \cdot \mathrm{v}^T{}_k \rangle \equiv Q_k$.

The conventional Kalman filter algorithm (details in Fig. 3) consists of three stages:

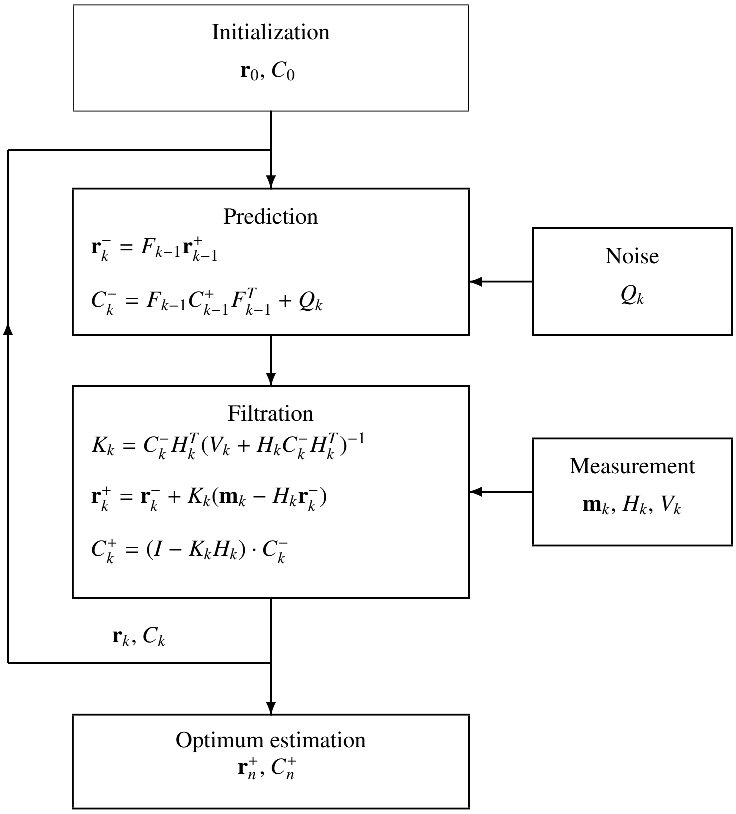

Fig. 3. Block diagram of the conventional Kalman filter.

Initialisation: The state vector **r** is initialised either arbitrary or with some approximate values. The covariance matrix is set to $C_0 = I \cdot \mathrm{inf}^2$, where inf denotes a large number.

Prediction: The current estimations of the state vector and the covariance matrix at the measurement $\mathbf{m}_{k-1}$ are propagated to the next measurement, and the process noise is taken into account. For the first propagation the initialisation values are used instead of a non-existed measurement.

Filtration: The predicted state vector and the covariance matrix are updated with the new measurement to get their optimal estimations, also at this stage we calculate $\boldsymbol{\zeta}_k$ – the residual, distance between the predicted and the actual measurement and W_k – the weight matrix, inverse covariance matrix of the residual:

$$\zeta_k = \mathbf{m}_k - H_k\,\mathbf{r}_k,$$

$$W_k = (V_k + H_k C_k^- H_k^T)^{-1}.$$

The following designations have been used: $\mathbf{r}^+{}_{k-1}$, C^+ – the optimum estimation and the error covariance matrix, obtained at the previous measurement; the matrix F_{k-1} relates the state at step $k-1$ to the state at step k; $\mathbf{r}^-{}_k$, C_k^- – predicted estimation of r_t and covariance matrix after the process noise; $\mathbf{m}_k$, V_k – the k-th measurement and its covariance matrix; the matrix H_k – the model of measurement; the value $\chi^2{}_k$ is the total χ^2-deviation of the obtained estimation $\mathbf{r}^+{}_k$ from the measurements $\mathbf{m}_1,\ldots,\mathbf{m}_k$.

The vector $\mathbf{r}^+{}_n$ obtained after the filtration of the last measurement is the desired optimal estimation of the $\mathbf{r}^t{}_n$ with the covariance matrix C_n^+.

In the our exercise we will deal with simple example of straight line trajectory in 2D space. Fig. 4 shows five detector planes placed along x axis. The distance between neighbouring detectors is L. We measure y-coordinate of a track in each detector plane with some error σ: (y_1, y_2, y_3, y_4, y_5). The task is to estimate the trajectory in the area of track origin IP. We will start estimation procedure with the last station 5. In this case of straight track the equation of motion is:

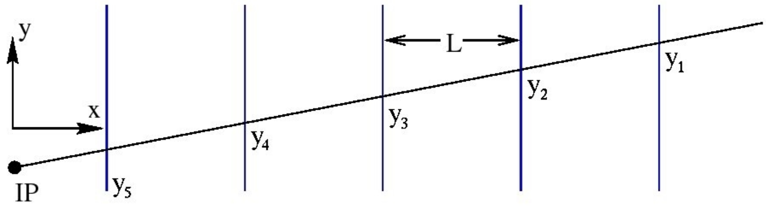

Fig. 4. Straight line track crossing detector planes.

$$y = t_y x + b.$$

Let us define state vector as the y-coordinate and tangent of track slope in x direction t_y and covariance matrix for this state vector:

$$\mathbf{r} = \begin{pmatrix} y \\ t_y \end{pmatrix}, \; C = \begin{pmatrix} C_{yy} & C_{yt_y} \\ C_{yt_y} & C_{t_y t_y} \end{pmatrix}.$$

Since we measure only y-coordinate the measurement vector for this case and its' covariance matrix are:

$$\mathbf{m}_k = \{y_k^m\}, \; V_k = \sigma^2.$$

The model of measurement for this case: $H_k = \begin{pmatrix} 1 & 0 \end{pmatrix}.$

So that: $y_k = H_k r_k$.

The propagation operator will be:

$$F_{k-1} = \begin{pmatrix} \frac{\partial y_k^-}{\partial y_{k-1}^+} & \frac{\partial y_k^-}{\partial t_{y_{k-1}}^+} \\ \frac{\partial t_{y_k}^-}{\partial y_{k-1}^+} & \frac{\partial t_{y_k}^-}{\partial t_{y_{k-1}}^+} \end{pmatrix} = \begin{pmatrix} 1 & -L \\ 0 & 1 \end{pmatrix}$$

So that the prediction stage can be rewritten in a way:

$y_k = y_{k-1} - L\, t_{yk}$.

$t_{yk} = t_{yk-1}$.

For the next vectorisation exercise we will have a closer look at KF algorithm in the case of straight line trajectory. The task is to SIMDise the estimation of particle tracks parameters using Kalman filter method[1]. The program consists of two parts: 1. simulation of particle tracks and 2. reconstruction of particle tracks parameters. An independent classes **LFSimulator** and **LFFitter** are present respectively for each task, both classes contain the parameters information of track environment, procedures to change those parameters and procedures to execute the task.

Only the reconstruction part must be vectorized. It is proposed to use templates for convenience of vectorisation and debugging, i.e. to create template classes and functions, which can be applied both to scalar and simd variables.

LFSimulator basing on given parameters of particle trajectory simulates interaction points with detector planes (Monte Carlo (MC) points) and detector measurements obtained due to these interactions (hits), they both structured into **LFTrack** class.

---

[1] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth and W.F.J. Müller, Fast SIMDized Kalman filter based track fit. CBM-SOFT-note-2007-001, 22 January 2007.

Part of the source code of KFLineFitter.cpp

```
struct LFPoint {
  LFPoint():x(NAN0),z(NAN0){};
  LFPoint( float x_, float z_ ): x(x_),z(z_) {};

  float x; // x-position of the hit
  float z; // coordinate of station
};
…
struct LFTrack {
  vector<LFPoint> hits;

  LFTrackParam rParam; // track parameters reconstructed by the fitter
  LFTrackCovMatrix rCovMatrix; // error (or covariance) matrix
  float chi2; // chi-squared deviation between points and trajectory
  int ndf;    // number degrees of freedom

  vector<LFTrackParam> mcPoints; // simulated track parameters
};
```

LFFitter basing on hits reconstructs parameters of particle trajectory and their error matrices (covariance matrices), chi-squared deviation between points and trajectory and number of degrees of freedom (NDF), which are also kept in **LFTrack** class.

Part of the source code of KFLineFitter.cpp

```
struct LFTrackParam {
…
  float &X() { return p[0]; };
  float &Tx() { return p[1]; };
  float &Z() { return z; };

  float p[2]; // x, tx.
  float z;
…
};
…
void LFFitter::Fit( LFTrack& track ) const
{
  Initialize( track );

  const int NHits = track.hits.size();
  for ( int i = 0; i < NHits; ++i ) {
    Extrapolate( track, track.hits[i].z );
    Filter( track, track.hits[i].x );
  }

  Extrapolate( track, track.mcPoints.back().z ); // exptrapolate to MC point for comparison with MC info
}
```

Part of the source code of KFLineFitter.cpp

```
void LFFitter::Initialize( LFTrack& track ) const
{
  track.rParam.Z() = 0;
  track.rParam.X() = 0;
  track.rParam.Tx() = 0;
  track.chi2 = 0;
  track.ndf = -2;

  track.rCovMatrix.C00() = InfX;
  track.rCovMatrix.C10() = 0;
  track.rCovMatrix.C11() = InfTx;
}

void LFFitter::Extrapolate( LFTrack& track, float z_ ) const
{
  float &z = track.rParam.Z();
  float &x = track.rParam.X();
  float &tx = track.rParam.Tx();
  float &C00 = track.rCovMatrix.C00();
  float &C10 = track.rCovMatrix.C10();
  float &C11 = track.rCovMatrix.C11();

  const float dz = z_ - z;

  x += dz * tx;
  z = z_;

  // F = 1  dz
  //     0  1

  const float C10_in = C10;
  C10 += dz * C11;
  C00 += dz * ( C10 + C10_in );
}

void LFFitter::Filter( LFTrack& track, float x_ ) const
{

  float &x = track.rParam.X();
  float &tx = track.rParam.Tx();
  float &C00 = track.rCovMatrix.C00();
  float &C10 = track.rCovMatrix.C10();
  float &C11 = track.rCovMatrix.C11();

    // H = { 1, 0 }
    // zeta = Hr - r // P.S. not "r - Hr" here becase later will be rather "r = r - K * zeta" then  "r = r + K * zeta"
  float zeta = x - x_;

    // F = C*H'
```

Part of the source code of KFLineFitter.cpp

```
239   float F0 = C00;
240   float F1 = C10;
241
242     // H*C*H'
243   float HCH = F0;
244
245     // S = 1. * ( V + H*C*H' )^-1
246   float wi = 1./( fSigma*fSigma + HCH );
247   float zetawi = zeta * wi;
248
249   track.chi2 += zeta * zetawi ;
250   track.ndf  += 1;
251
252     // K = C*H'*S = F*S
253   float K0 = F0*wi;
254   float K1 = F1*wi;
255
256     // r = r - K * zeta
257   x  -= K0*zeta;
258   tx -= K1*zeta;
259
260     // C = C - K*H*C = C - K*F
261   C00 -= K0*F0;
262   C10 -= K1*F0;
263   C11 -= K1*F1;
264
265 }
```

2_SIMD/3_KF: solution

First of all it has to be decided which data should be grouped and how it should be grouped to vectorize the track fitting procedure. The grouped data should be maximally independent, therefore the most simple and effective way is to treat M (4) independent tracks in parallel. The procedure (see lines 177-188 in scalar version) would be the following: M tracks are initialised, M tracks are extrapolated to the 1-st station, M hits are taken into account in the tracks parameters estimation (one hit per track), M tracks extrapolated to 2-nd station, … M tracks extrapolated to z-coordinate of last mc point, which must be the same for all tracks.

To perform these procedures we should prepare hits grouping them from different tracks into one vector, all hits in group must be on the same station. Corresponding class should have vector of M x-coordinates of M hits and scalar of z-coordinate of M hits, which is same as z-coordinate of a station the hits belong to. The general type for M **float**s grouped together is noted as **T**. Both **fvec** and **float** types can be substituted here instead of **T**, that justifies the template construct usage.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
template< typename T >
struct LFPoint {
  LFPoint():x(NAN0),z(NAN0){};
  LFPoint( T x_, T z_ ): x(x_),z(z_) {};

  T x; // x-position of the hit
```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
float z; // coordinate of station // all points on one station have same z-position
};
```

Result of the procedure would be M track parameters grouped into one vectorized parameters class. Similarly to hits, **x** and **Tx** parameters and covariance elements are grouped together and z-coordinate is stays scalar.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
template< typename T >
struct LFTrackParam {
…
  T &X() { return p[0]; };
  T &Tx() { return p[1]; };
  float &Z() { return z; };

  T p[2]; // x, tx.
  float z;
…
};

template< typename T >
struct LFTrackCovMatrix {
  T &C00() { return c[0]; };
  T &C10() { return c[1]; };
  T &C11() { return c[2]; };

  T c[3]; // C00, C10, C11
…
};
```

The data is grouped in track class, which also have additional chi-squared deviation and NDF, which can be different for different tracks, therefore required a vector type. Meanwhile NDF is integer, therefore additional parameter of template **I** is added for grouped integers.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
template< typename T, typename I >
struct LFTrack {
  vector< LFPoint<T> > hits;

  LFTrackParam<T> rParam; // reconstructed by the fitter track parameters
  LFTrackCovMatrix<T> rCovMatrix; // error (or covariance) matrix
  T chi2;    // chi-squared deviation between points and trajectory
  I ndf;     // number degrees of freedom

  vector< LFTrackParam<T> > mcPoints; // simulated track parameters
};
```

The same operations must be done in LFFitter functions, which implement data processing: basically all floats, with exception of z-coordinate, should be changed to template **T** type and all integers to **I** type, and class types to the templates prepared for vector processing. Since 4 tracks are independent and similar no changes in the algorithm itself are required and the code is basically the same.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
template< typename T, typename I >
void LFFitter<T,I>::Fit( LFTrack<T,I>& track ) const
{
  Initialize( track );
  const int NHits = track.hits.size();
  for ( int i = 0; i < NHits; ++i ) {
    Extrapolate( track, track.hits[i].z );
    Filter( track, track.hits[i].x );
  }

  Extrapolate( track, track.mcPoints.back().z ); // just for pulls
}

template< typename T, typename I >
void LFFitter<T,I>::Initialize( LFTrack<T,I>& track ) const
{
  track.rParam.Z() = 0;
  track.rParam.X() = 0;
  track.rParam.Tx() = 0;
  track.chi2 = 0;
  track.ndf = -2;

  track.rCovMatrix.C00() = InfX;
  track.rCovMatrix.C10() = 0;
  track.rCovMatrix.C11() = InfTx;
}

template< typename T, typename I >
void LFFitter<T,I>::Extrapolate( LFTrack<T,I>& track, float z_ ) const
{
  float &z = track.rParam.Z();
  T &x = track.rParam.X();
  T &tx = track.rParam.Tx();
  T &C00 = track.rCovMatrix.C00();
  T &C10 = track.rCovMatrix.C10();
  T &C11 = track.rCovMatrix.C11();

  const float dz = z_ - z;

  x += dz * tx;
  z = z_;

  // F = 1  dz
  //     0   1

  const T C10_in = C10;
  C10 += dz * C11;
```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
  C00 += dz * ( C10 + C10_in );
}

template< typename T, typename I >
void LFFitter<T,I>::Filter( LFTrack<T,I>& track, T x_ ) const
{

  T &x = track.rParam.X();
  T &tx = track.rParam.Tx();
  T &C00 = track.rCovMatrix.C00();
  T &C10 = track.rCovMatrix.C10();
  T &C11 = track.rCovMatrix.C11();

    // H = { 1, 0 }
    // zeta = Hr - r // P.S. not "r - Hr" here becase later will be rather "r = r - K * zeta" then  "r = r + K * zeta"
  T zeta = x - x_;

  // F = C*H'
  T F0 = C00;
  T F1 = C10;

    // H*C*H'
  T HCH = F0;

    // S = 1. * ( V + H*C*H' )^-1
  T wi = 1./( fSigma*fSigma + HCH );
  T zetawi = zeta * wi;

  track.chi2 += zeta * zetawi ;
  track.ndf  += 1;

    // K = C*H'*S = F*S
  T K0 = F0*wi;
  T K1 = F1*wi;

    // r = r - K * zeta
  x  -= K0*zeta;
  tx -= K1*zeta;

    // C = C - K*H*C = C - K*F
  C00 -= K0*F0;
  C10 -= K1*F0;
  C11 -= K1*F1;

}
```

All the template classes and functions can be used for scalar calculations in the same way as before, just adding **<float,int>** template parameters to fit class:

Part of the source code of KFLineFitter_solution2_simd.cpp

```
#ifndef SIMDIZED

  LFFitter<float,int> fit;

  fit.SetSigma( Sigma );

#ifdef TIME
  timer.Start(1);
#endif
  for ( int i = 0; i < NTracks; ++i ) {
    LFTrack<float,int> &track = tracks[i];
    fit.Fit( track );
  }
  #ifdef TIME
  timer.Stop();
  #endif

#else
```

The **LFFitter** class can be used similarly for vectored computations, **T** parameter should be set to **fvec**, **I** parameter should be set to **fvec** either, since we can use floating point values to store integers. In addition the input data should be prepared and the output data should be converted to scalar format for future comparison. For this purpose one should introduce two additional functions: **CopyTrackHits** and **CopyTrackParams**.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
#else

    // Convert scalar Tracks to SIMD-tracks
  const int NVTracks = NTracks/fvecLen;
  LFTrack<fvec,fvec> vTracks[NVTracks];

  CopyTrackHits( tracks, vTracks, NVTracks );

    // fit
  LFFitter<fvec,fvec> fit;

  fit.SetSigma( Sigma );

#ifdef TIME
  timer.Start(1);
#endif
  for ( int i = 0; i < NVTracks; ++i ) {
    LFTrack<fvec,fvec> &track = vTracks[i];
    fit.Fit( track );
  }
#ifdef TIME
  timer.Stop();
#endif

```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
402     // Convert  SIMD-tracks to scalar Tracks
403   CopyTrackParams( vTracks, tracks, NVTracks );
404
405 #endif // SIMDIZED
```

The **CopyTrackHits** function is needed to copy all required by **LFFitter** class data into vectorized classes. These are full hits data and z-coordinate of the last MC point. To copy it one would need a loop over groups of tracks, **fvecLen** tracks in group (see line 384). For each group loop over track in group are required and a loop over hits in track (lines 111 and 114). Since all tracks have same number of hits, equal to number of stations we can take this number from the very first track and make it constant. The z-coordinate of the last point should be copied for each track after loop over hits.

The **CopyTrackParams** function is needed to copy all output data from vectorized classes to scalar classes. This would require similarly the loop over vectorized tracks, and loop over entries in the vectorized tracks, loop over parameters and loop over covariance matrix elements.

Part of the source code of KFLineFitter_solution2_simd.cpp

```
103 void CopyTrackHits( const LFTrack<float,int>* sTracks, LFTrack<fvec,fvec>* vTracks, int nVTracks ){
104   const int NHits = sTracks[0].hits.size(); // all tracks have the same number of hits
105
106
107   for( int iV = 0; iV < nVTracks; ++iV ) {
108     LFTrack<fvec,fvec>& vTrack = vTracks[iV];
109     vTrack.hits.resize(NHits);
110     vTrack.mcPoints.resize(NHits);
111     for( int i = 0; i < fvecLen; ++i ) {
112       const LFTrack<float,int>& sTrack = sTracks[ iV*fvecLen + i ];
113
114       for( int iH = 0; iH < NHits; ++iH ) {
115         vTrack.hits[iH].x[i] = sTrack.hits[iH].x;
116         vTrack.hits[iH].z = sTrack.hits[iH].z;
117       }
118
119       vTrack.mcPoints[NHits-1].z = sTrack.hits[NHits-1].z; // need this info for comparison of reco and MC
120     }
121   }
122 }
123
124 void CopyTrackParams( const LFTrack<fvec,fvec>* vTracks, LFTrack<float,int>* sTracks, int nVTracks ){
125
126
127   for( int iV = 0; iV < nVTracks; ++iV ) {
128     const LFTrack<fvec,fvec>& vTrack = vTracks[iV];
129     for( int i = 0; i < fvecLen; ++i ) {
130       LFTrack<float,int>& sTrack = sTracks[ iV*fvecLen + i ];
131
132       for( int ip = 0; ip < 2; ++ip )
133         sTrack.rParam.p[ip] = vTrack.rParam.p[ip][i];
134       sTrack.rParam.z = vTrack.rParam.z;
```

Part of the source code of KFLineFitter_solution2_simd.cpp

```
      for( int ic = 0; ic < 3; ++ic )
        sTrack.rCovMatrix.c[ic] = vTrack.rCovMatrix.c[ic][i];
      sTrack.chi2 = vTrack.chi2[i];
      sTrack.ndf = vTrack.ndf[i];
    }
  }
}
```

The finial output, saved in the file must be the same for scalar and vector version. The time should be about factor 4 different.

Typical output of KFLineFitter.cpp

```
 Begin
Time: 2.35605 ms
 End
```

Typical output of KFLineFitter_solution2_simd.cpp

```
 Begin
Time: 0.647068 ms
 End
```