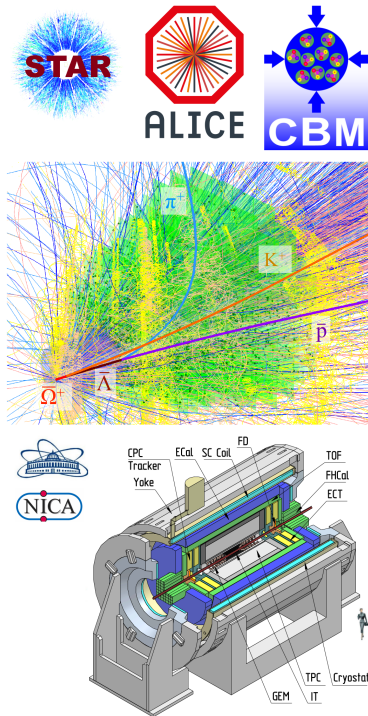


# Use of KFParticle with MPD



## Contacts:

Pavel Batyuk, VBLHEP, JINR, [pavel.batyuk@jinr.ru](mailto:pavel.batyuk@jinr.ru)

# Contents

<b>Contents</b>	<b>I</b>
1 Introduction . . . . .	1
2 How to produce output for analysis? . . . . .	2
3 How to read produced output? . . . . .	7
4 The most useful methods provided by <b>KFParticle</b> to be used in user's analysis . . . . .	11

# 1 Introduction

KFParticle is a formalism specially developed for complete reconstruction of short-lived particles with their momentum  $P$ , energy  $E$ , mass  $m$ , lifetime  $c\tau$ , decay length  $L$  and so on. Its main benefits are listed below:

- based on the Kalman filter mathematics independent in sense of experimental setup (collider or fixed target)
- allows one reconstruction of decay chains (cascades)
- daughter and mother particles are described and considered the same way
- daughter particles are added to the mother particle independently

At the moment, the power abilities of KFParticle can be used in the MpdRoot software. It is embedded into MpdRoot as a module with all necessary dependencies and a first version of particle finder respectively to MPD that uses all the benefits by KFParticle, has been developed. The manual has a goal to share experience on how to use the main macroses aiming at production of output with its subsequent analysis in user's codes. It is not a finished job since the development (description) procedure is ongoing.

## 2 How to produce output for analysis?

Right now, the package has one main macro to be used for starting the software (namely, to produce output).

The macro can be found in the [macro/KFParticle/runKFParticle](#) directory of the MPD software. It is called [runKFParticleFinder.C](#) (see below in p. 6).

It inherits all benefits from FairRoot to work in a “task”-way mode processing passed events in chain.

There are two options how user can pass input data to the macro. The most trivial one is that one passes a single file written either in the [standard dst-format](#) or in the [MiniDst format](#). The latter is fully supported by the software, but one has to take into account that MiniDst input should be produced with a code version where there is a correct parameterization of covariance matrix used. Looking at a “suitable” file one sees the following parameterization of covariance matrix:

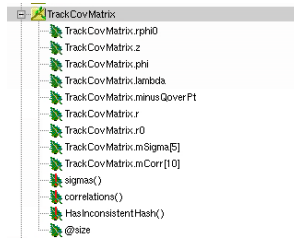


Figure 0.1: **Example how a correct parameterization of covariance matrix looks like**

The second option how to provide input for the software looks more preferred since it works with a list of files. The passed list should have \*.lis or \*.list extension and the following format:

```

0 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-13-.reco.MinIDst.root
1 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-10-.reco.MinIDst.root
2 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-2-.reco.MinIDst.root
3 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-3-.reco.MinIDst.root
4 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-5-.reco.MinIDst.root
5 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-11-.reco.MinIDst.root
6 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-1-.reco.MinIDst.root
7 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-20-.reco.MinIDst.root
8 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-9-.reco.MinIDst.root
9 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-4-.reco.MinIDst.root
10 dcmsmm-BiBi-09.2GeV-mb-eos0-2k-7-.reco.MinIDst.root

```

**Figure 0.2: Example of list (\*.lis or \*.list) to be passed to software**

Each line contains a file identifier (file ID) and dst name (supported, surely, both formats as mentioned before). The identifier is considered to be obligatory since it allows one in subsequent analysis to get information on file name and, in particular, event number. The list is limited by five thousand (5000) lines (e.g. files) to be put inside.

Also user should define an output file name and desirable number of events to process. The number of events in the case of a passed list means number of consecutive events (from the total number of events passed in the list) aimed to be processed. If the number is equal to zero, it means that the whole dataset is assumed to be processed.

The macro also instantiates the Kalman filter since it is used for doing extrapolation of track parameters to the inner TPC shell.

The main object of the software is called **MpdKfParticleFinder** and represents itself a first **version of particle finder approach based on the KFParticle formalism respectively to MPD**. Generally speaking, it is an interface to KFParticle that takes into account important things how to work with existing reconstruction output from the detector (namely, TPC at the moment). It does support of two existing dst formats in MPD and used track parameterization (see Fig.0.3).

$$\begin{cases} x = r \cos \phi_t \\ y = r \sin \phi_t \\ z = z \\ P_x = |P_t| \cos \phi \\ P_y = |P_t| \sin \phi \\ P_z = |P_t| \tan \lambda \end{cases}$$

Track params. are given at  
2D DCA point to beamline  
(X, Y = 0)

Track state vector =  $\begin{pmatrix} r\phi_t \\ z \\ \phi \\ \lambda = \pi/2 - \theta \\ -q/P_t \end{pmatrix}$

$C_{out} = J C_{in} J^T$

$$J_{2-out} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -|P_t| \sin \phi & 0 & \frac{p_t^2 \cos \phi}{q} \\ 0 & 0 & 0 & |P_t| \cos \phi & 0 & \frac{p_t^2 \sin \phi}{q} \\ 0 & 0 & 0 & 0 & \frac{|P_t|}{\cos^2 \lambda} & \frac{p_t^2 \tan \lambda}{q} \end{pmatrix}$$

Required extrapolation of  
track params. to the TPC  
inner shell (first hit position)

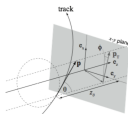


Figure 0.3: **Example of track parameterization in MPD**

The latter is the most important part of the particle finder. Also, a simple mechanism to select input tracks with desirable kinematical characteristics (transverse momentum, rapidity) is realized.

A brief explanation of the main modifiers to **MpdKfParticleFinder** is given in the table. The modifiers are set by

## 2. HOW TO PRODUCE OUTPUT FOR ANALYSIS? 5

default values in the software, but user has a possibility to reset them if necessary.

Modifier	Explanation
<b>SetPidHypo(pdg1, pdg2)</b>	<p><b>Default values are 2212 and -211</b></p> <p>At the moment, it is possible to select only one additional decay for analysis. Setting (+211, -211) corresponds to <math>K_s^0 \rightarrow \pi^+ + \pi^-</math></p> <p><b>Temporary solution to change in the future</b></p>
<b>SetUseCuts(flag)</b>	<p><b>Default flag value is false.</b></p> <p>If set with true, it means that tracks are chosen according to the established values. The cuts are set by <b>TrackCuts</b>. At the moment, it is possible to define values for pseudorapidity and transverse momentum in sense of track acceptance.</p>
<b>SetUseKFPerformance(flag)</b>	<p><b>Default flag value is false.</b></p> <p>Returning it to true gives opportunity to produce QA histograms proposed by KFPparticle performance tools. They seem to be very useful when doing comparison between Monte Carlo and reconstruction aiming at developing cuts to suppress background and so on.</p> <p><b>Important note: it works just with standard dst!!!</b></p>

```

void runKFParticleFinder(TString in = "", TString out = "",
Int_t nStartEvent = 0, Int_t nEvents = 0) {
FairRunAna* fRun = new FairRunAna();
FairRootFileSink* sink = new FairRootFileSink(out);
fRun->SetSink(sink);

MpdKalmanFilter *kalman = MpdKalmanFilter::Instance("KF");
fRun->AddTask(kalman);

MpdKfParticleFinder* pF = new MpdKfParticleFinder(in);
// pF->SetUseCuts(kTRUE);
// pF->SetUseKFPerformance(kTRUE);

// Setters to specify what a two-particle decay we are looking for ...
// pdg1, pdg2 to specify a decay. \Lambda^{0} is set by default
// Available decays right now are:
// \Lambda^{0} -> p + \pi^{-} and
// \K^{0}_{S} -> \pi^{+} + \pi^{-}

// pF->SetPidHypo(+211, -211);

fRun->AddTask(pF);
fRun->Init();

// Redefine established cuts if necessary ...
//   TrackCuts* cutsGot = partFinder->GetTrackCuts();
//   if (cutsGot) {
//       cutsGot->SetPtMin(.5);
//       cutsGot->SetPtMax(1.8);
//       cutsGot->SetAbsEta(1.);
//   }

fRun->Run(nStartEvent, nStartEvent + nEvents);

delete pF;
}

```



### 3 How to read produced output?

To use a result of work of particle finder one is referred to [readKFParticleOutput.C](#) located, evidently, in the macro/KF-Particle directory. The macro illustrates how to get output from the previous step and use it.

Besides of decaying particles to be reconstructed, the software is able to perform reconstruction of primary vertex ( $V_p$ ) of interaction. In the most general case, it allows (KFParticle) one to get more than one primary vertex reconstructed, but existing current setup of the software reconstructs one candidate to be primary vertex. At the moment, multiple vertices are not allowed to be reconstructed (particle finder).

Below are given explanations for some parts of [readKFParticleOutput.C](#) considered to be useful for users in their analyses.

```
...
// Getting information on reco file and event used ...
// id of file in the list processed by runKFParticle.C
Int_t fileId = eventHeader->GetInputFileId();

// current event that corresponds to the same event in reco output
Int_t evNum = eventHeader->GetMCEntryNumber();
...
```

A couple of values ([fileId](#), [evNum](#)) are important when linking output from the software to the source reconstructed data. [fileId](#) corresponds to those one defined in the list of files passed for analysis. So, taking into account this pair, one is able to get event ([evNum](#)) with all information from the source.

Information on reconstructed primary vertex is written into output with [KFVertex](#)

```

...
// Getting reconstructed primary vertex in event ...
// In general case, here should be a loop since we are able to reconstruct
// more than one primary vertex ...
// One vertex is set by default in macro/KFParticle/runKFParticleFinder.C
// It means that primVertices->GetEntriesFast() is equal to 1

for (Int_t iVertex = 0; iVertex < primVertices->GetEntriesFast(); iVertex++) {
  KFVertex* Vp = (KFVertex*) primVertices->UncheckedAt(iVertex);

  // To get a list of all possible accessors to written data, please,
  // look at KFParticle/KFParticle/KFVertex.h

  Float_t* covMatrix = Vp->CovarianceMatrix();

  // Getting components of reconstructed vertex ...
  Double_t X = Vp->GetX();
  Double_t Y = Vp->GetY();
  Double_t Z = Vp->GetZ();

  // X = Y = Z = 0. means that primary vertex
  // has not been reconstructed in event ...
  if (!X || !Y || !Z)
    continue;

  Int_t nContributors = Vp->GetNContributors();
}
...

```

The most important information that can be derived from the vertex section is three-dimensional coordinates of reconstructed vertex, number of participants used for the vertex reconstruction and, surely, covariance matrix of vertex fit. A full list of accessors to be used is available at **KFParticle/KFParticle/KFVertex.h**. All necessary brief explanations are given there directly in the code when describing accessors.

Reconstructed secondary candidates are written to output with a very widely and frequently used **KFParticle** inherited from the **KFParticleBase** class.

A cycle illustrating how to get information on reconstructed

secondaries is given below. To select candidates that, probably, are considered as particles we are interested in, one uses PDG hypothesis of interesting particle. So, to get, for example,  $\Lambda^0$  candidates one has to do as follows:

```
for (Int_t iSecondary = 0; iSecondary < recoSecondaries->GetEntriesFast(); iSecondary++) {
    KFPParticle* particle = (KFPParticle*) recoSecondaries->UncheckedAt(iSecondary);

    // To get a list of all possible accessors to written data, please,
    // look at KFPParticle/KFPParticle/KFPParticle.h and KFPParticle/KFPParticle/KFPParticleBase.h

    Int_t pdgHypo = particle->GetPDG();
    if (pdgHypo != pdg)
        continue;

    Double_t mass = particle->GetMass();

    Double_t Px = particle->GetPx();
    Double_t Py = particle->GetPy();
    Double_t Pz = particle->GetPz();

    Double_t pT = particle->GetPt();
    Double_t eta = particle->GetEta();
}
```

**KFPParticle** has a lot of accessors to parameters of reconstructed secondaries. It is possible to get **mass, projection of momentum, rapidity, decay length** and many other characteristics of found secondaries. A full list of accessors with brief descriptions is given in **KFPParticle/KFPParticle/KFPParticle.h** and **KFPParticle/KFPParticle/KFPParticleBase.h**. The most useful of them are planned to be marked and explained in Section 4 in the nearest future.

Another important option is related to possibility of getting indices of particles that form interesting decay channel of decaying particle. It is illustrated below how to do it. The feature works correctly just in case of standard dst used, since this case (at the moment, MiniDst does not support it yet) we are able

to construct a full chain that allows one to pass through all necessary steps and, finally, to get index of mother particle to judge whether the found candidate is true or false assuming to be a desirable decaying particle.

```
// It is possible to get indices of reconstructed tracks in standard / MiniDst.
// Doing it for standard dst, one can get source Monte Carlo track to check
// whether its mother is a desirable decaying particle we are looking for ...
// (daughter id given by particle finder --> idx of reconstructed track -->
// --> idx of corresponding MC track --> idx of mother particle)
// To do it, one has to link standard dst to the macro in an usual way
// fileId and evNum are available to get corresponding file, event and arrays
// with reconstructed tracks

const Double_t massThresh = 0.025;
if (TMath::Abs(mass) < massThresh) {
    Int_t nDaugh = particle->NDaughters();
    vector <Int_t> daughters = particle->DaughterIds();

    for (Int_t iDaugh = 0; iDaugh < nDaugh; iDaugh++)
        cout << daughters[iDaugh] << " ";
    cout << endl;
}
```

#### 4 The most useful methods provided by **KFParticle** to be used in user's analysis

The section is planned to be filled with descriptions of useful methods on how to use them correctly in user's analysis.