Tracking

Lia Lavezzi
INFN Pavia

Summary

- ❖ Introduction
- ❖ Input/Output Data
- ❖ Pattern Recognition
- ❖ Track Fitting
- ❖ Track Extrapolation
- ❖ Some macros…

Acknowledgements

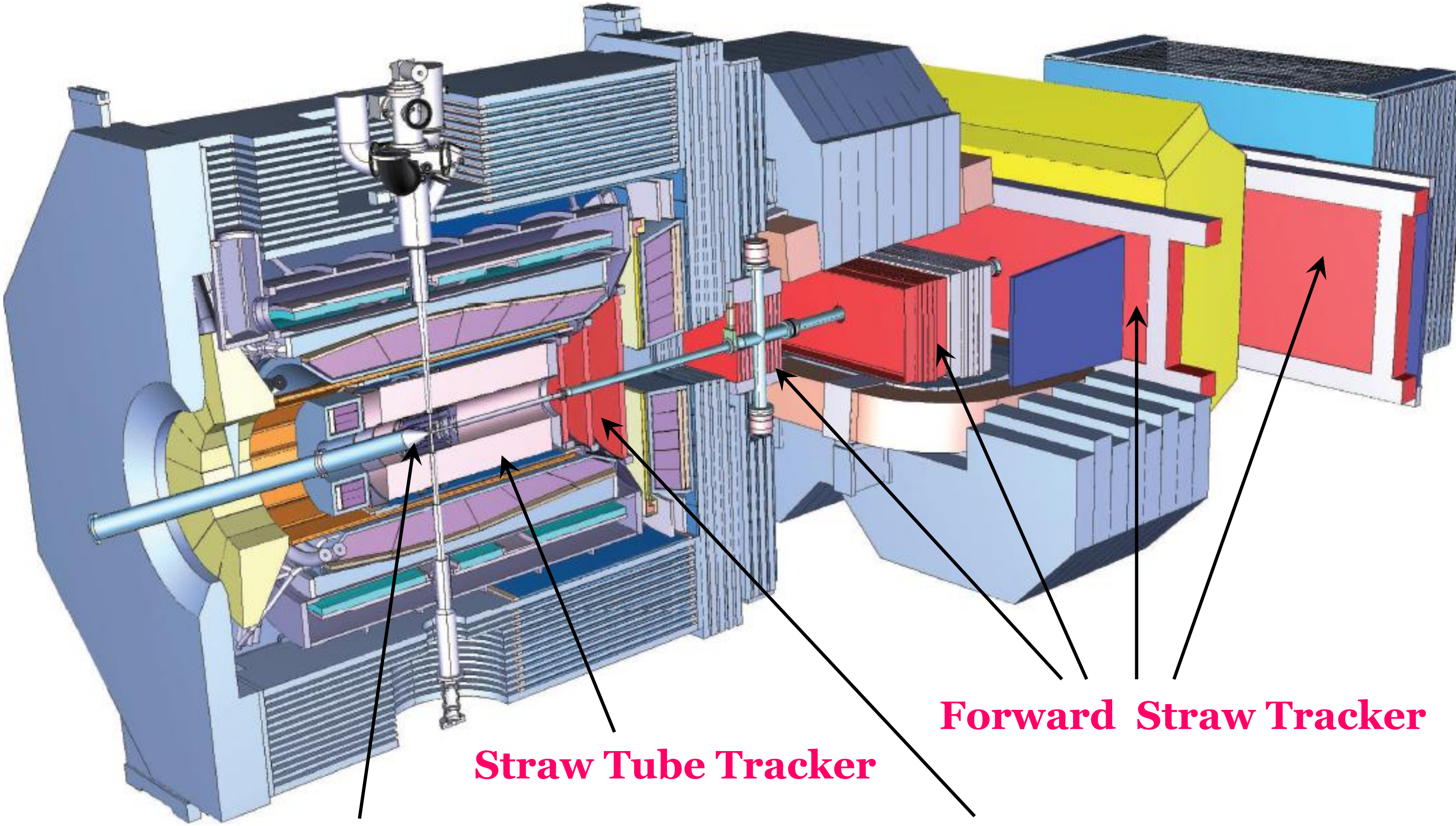
Many thanks to everyone who will recognize her/his contribution in this talk, in terms of slides, pictures, comments (yes, I "stole" from *your* past presentations here and there) and everyone who helped me answering my questions during the preparation!

Summary

- Introduction
- Input/Output Data
- Pattern Recognition
- Track Fitting
- Track Extrapolation
- Some macros…

Introduction – what is the aim?

❖ The trackers must provide the positions where the particles crossed the detectors



Introduction – what is the aim?

❖ The hits left by energy depositions in the different detectors are taken as **input** for the track reconstruction

❖ They are like "footsteps" which indicate the particle trail

❖ They must be **grouped** to identify the original tracks

❖ The duty becomes more and more challenging as the number of tracks (→ hits) grows

Introduction – the procedure

The **full reconstruction** of a track is fulfilled through the following steps:

I. Association of the hits together in tracks	**PATTERN RECOGNITION**
II. Evaluation of the track parameters with a preliminary fit	**PREFIT**
III. Refinement of the parameters with a fitting procedure	**KALMAN FIT**
IV. Extrapolation of the parameters to the plane where we need to know them	**EXTRAPOLATION**

Summary

- ❖ Introduction
- ❖ Input/Output Data
- ❖ Pattern Recognition
- ❖ Track Fitting
- ❖ Track Extrapolation
- ❖ Some macros…

The Data

- The following objects *contain* the tracking information
- They “live” in `pnddata/TrackData`

- PndTrackCand
- PndTrack
- PndTrackCandHit
- PndTrackID

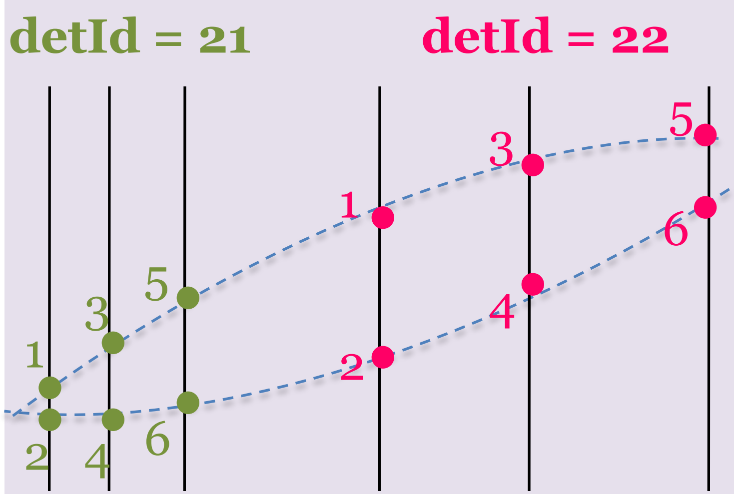
PndTrackCand

This is the **list of hits**

❖ Each hit is identified by the detId (detector identifier) and the hitId (hit idenfier)

❖ The detId identifies which detector collected the hit and which TClonesArray contains it

❖ The hitId corresponds to the index of the hit in the TClonesArray



TClonesArray 21 contains hits 1, 2, 3, 4, 5, 6
TClonesArray 22 contains hits 1, 2, 3, 4, 5, 6

There are 2 tracks:

Track I (21, 1) (21, 3) (21, 5) (22, 1) (22, 3) (22, 5)
Track II (21, 2) (21, 4) (21, 6) (22, 2) (22, 4) (22, 6)

PndTrackCand

- **ctor/dtor**

```
PndTrackCand();
~PndTrackCand();
```

- **data members**

  the list of hits is an **STL vector** of PndTrackCandHits

```
std::vector<PndTrackCandHit> fHitId;
```

- **functions**

```
void AddHit(UInt_t detId, UInt_t hitId, Double_t rho);
```

• *detId* identifies the detector, i.e. the TClonesArray where the hit is stored
• *hitId* is the index of the hit inside the TClonesArray
• *rho* is the variable according to which the sorting is performed (*e.g. the distance)*

```
std::vector<PndTrackCandHit> GetSortedHits();
void Sort();
```

The hits inside the STL vector can be sorted using the std::sort function
WARNING! rho must be chosen accurately to get the correct ordering

PndTrackCand

❖**WARNING!** There are still some “seed” values which are no more used, still there for historical reasons

```
int fMcTrackId;
TVector3 fPosSeed;
TVector3 fDirSeed;
double fQoverPseed;
```

... and the corresponding functions

```
void setMcTrackId(int i)
void setTrackSeed(const TVector3& p,const TVector3& d,double qop)
int getMcTrackId()
TVector3 getPosSeed()
TVector3 getDirSeed()
double getQoverPseed()
```

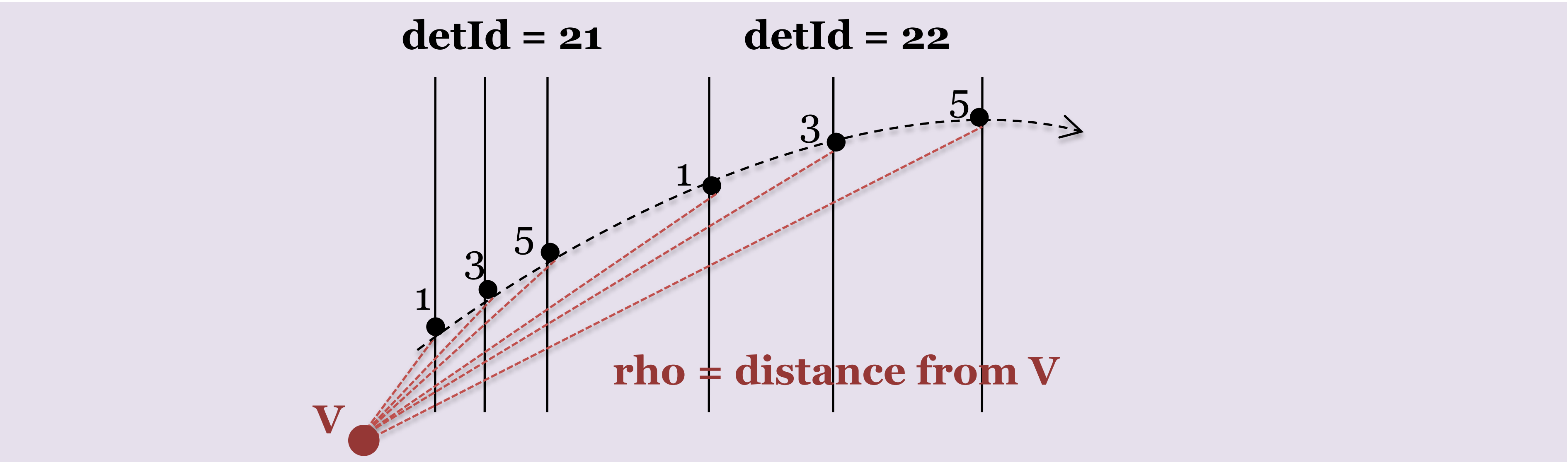
PndTrackCandHit

This is the **hit associated to the track**

❖ It is identified by the detId and the hitId

❖ It contains the parameter rho, according to which the hit list can be sorted

❖ When retrieving the hit from the PndTrackCand, it is a PndTrackCandHit: to retrieve the actual hit, you must get the detId (→ from which you choose the TCA) and the hitId (→ from which you get the hit inside the TCA)



PndTrackCandHit

- **ctor**

```
PndTrackCandHit(Int_t detId, Int_t hitId, Double_t rho)
```

- It inherits from **FairLink**, which contains the detId and hitId

```
class PndTrackCandHit : public FairLink
```

- in addition **rho** must be provided

```
Double_t fRho
```

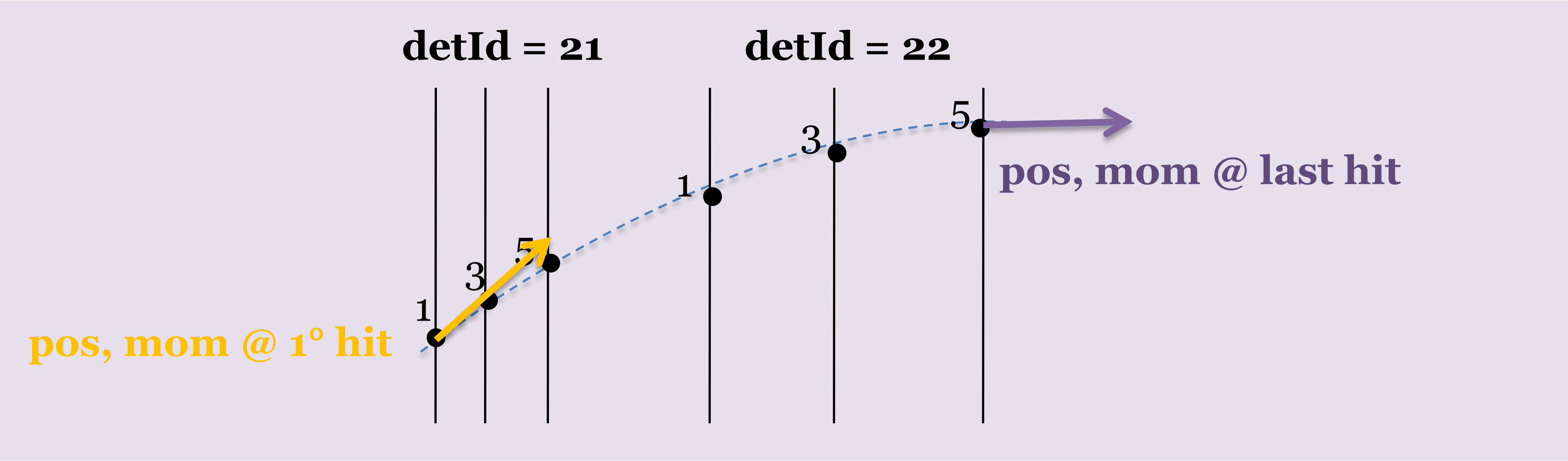
- the useful functions

```
Int_t GetHitId();
Int_t GetDetId();
Double_t GetRho();
```

PndTrack

This is the **hypothesis of track**

- The track is identified by the momentum and position at the first hit and the last hit
- The corresponding PndTrackCand is accessible from it



PndTrack

- **ctor**

```
PndTrack(const FairTrackParP& first, const FairTrackParP&
last, const PndTrackCand& cand, Int_t flag = 0, Double_t
chi2 = -1., Int_t ndf = 0, Int_t pid = 0, Int_t id = -1,
Int_t type = -1);
```

the main informations that must be provided to have a track are:

- the track parameters @ the 1st & last point, through the FairTrackParP object (*see later*)

```
FairTrackParP fTrackParamFirst;
FairTrackParP fTrackParamLast;
```

- and the corresponding list of hits, via the PndTrackCand object

```
PndTrackCand fTrackCand;
```

- A quality flag *(e.g. it says whether the fit is ok or not)*

```
Int_t fFlag;
```

PndTrackID

- It contains the **connections** of the reconstructed track **to the MC Track**

- **ctor**

```
PndTrackID(const Int_t id,
           const TArrayI track,
           const TArrayI mult)
```

- **class members**

```
Int_t   fTrackID;
TArrayI fCorrTrackIds;
TArrayI fMultTrackIds;
```

Reco track index in the TClonesArray
List of MC Track indices
List of multiplicities

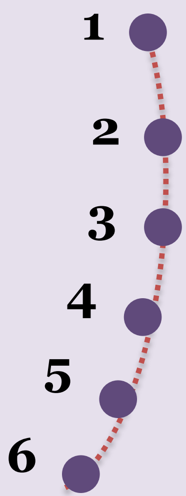
- function to retrieve the MC TrackIDs

```
Int_t GetCorrTrackID(Int_t i=0)
```

PndMCTrackAssociator

- it is a task to associate the reco track to one (or more) MC tracks
- it loops over all the PndTrackCandHit belonging to a PndTrackCand
- it goes from the PndTrackCandHit to the PndXXXHit to the PndXXXPoint
- it retrieves the MC track ID from the MC Point
- it counts how many hits belong to a certain MC track and fills the PndTrackID

example



CandHit	1	2	3	4	5	6
Hit	4	8	15	16	23	42
MC point	3	4	10	2	6	21
MC trackID	0	0	1	0	1	0

PndTrackID will say this reco track is connected to:
MCTrack 0, 4 times
MCTrack 1, 2 times

Data Summary

We have **two objects** to describe a track:

❖ PndTrackCand which is the list of hits
❖ PndTrack which contains the momentum and position of the track

Why two objects instead of one?
Because the same list of hits can be refitted with different algorithms, particle hypotheses, … and it makes no sense to copy several times the information on the list of hits, it is enough that every fitted track PndTrack contains the possibility to retrieve the corresponding PndTrackCand

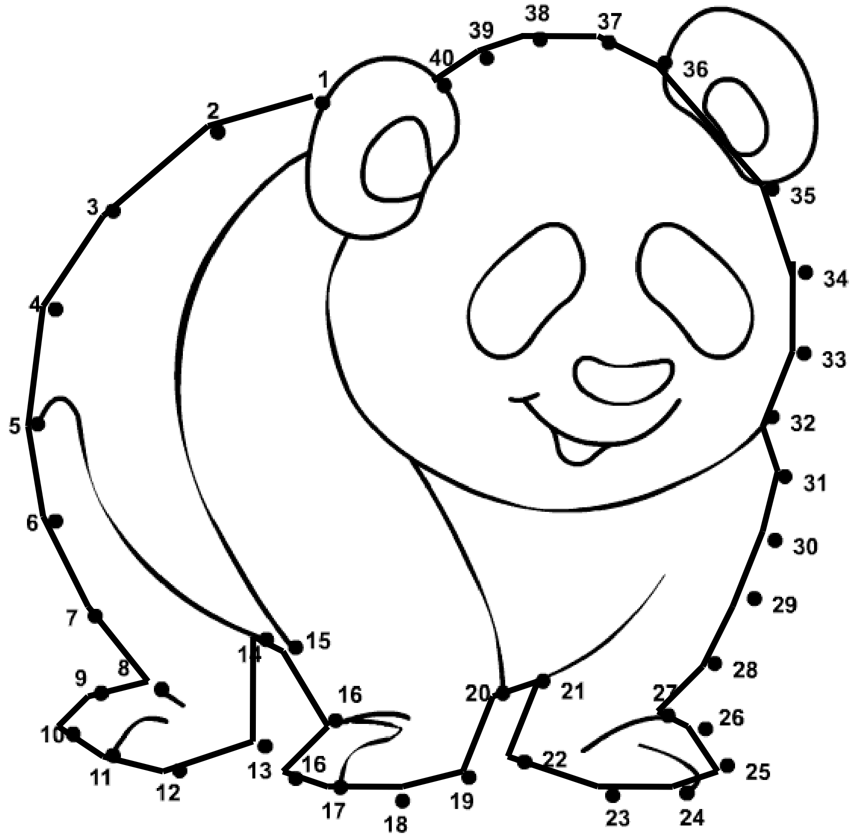
❖ We can access the hitId and detId from the PndTrackCand, via the PndTrackCandHit and from them we can pick the correct PndXXXHit (hitId) in the correct TClonesArray of the detector XXX (detId)

Summary

- ❖ Introduction
- ❖ Input/Output Data
- ❖ Pattern Recognition
- ❖ Track Fitting
- ❖ Track Extrapolation
- ❖ Some macros…

Pattern Recognition

It "connects the dots"



❖ It assigns the hits to the tracks they belong to

… but not only…

❖ The final step of the pattern recognition procedure is to provide a **first guess** of the track parameters: in a word, give the momentum and position of the track!

Pattern Recognition: prefit

❖ In the region where the highly **homogeneous** solenoidal field applies, i.e. in the target spectrometer trackers, it is enough to fit the hits with a helix, since the material budget is small as well as the field inhomogeneities.

❖ In the dipole region (and the transient field region, i.e. the region between the solenoid and the dipole), this is not possible anymore. Some other track representations are needed (spline?)

Pattern recognition

Input: the different TClonesArrays of hits from the different trackers

```
fXXXHitArray = (TClonesArray*) ioman->GetObject("XXXHit");
```

Output: the two TClonesArrays of PndTrack and PndTrackCand

```
fTrackCandArray = new TClonesArray("PndTrackCand");
ioman->Register("XXXTrackCand", "XXX", fTrackCandArray, kTRUE);

fTrackArray = new TClonesArray("PndTrack");
ioman->Register("XXXTrack", "XXX", fTrackArray, kTRUE);
```

In pandaroot we have different pattern recognitions:

1. **local**, for each tracking detector
2. **global**, which gathers information from different detectors
3. **ideal**, which takes the information from the Monte Carlo truth

Some examples of PR in pandaroot

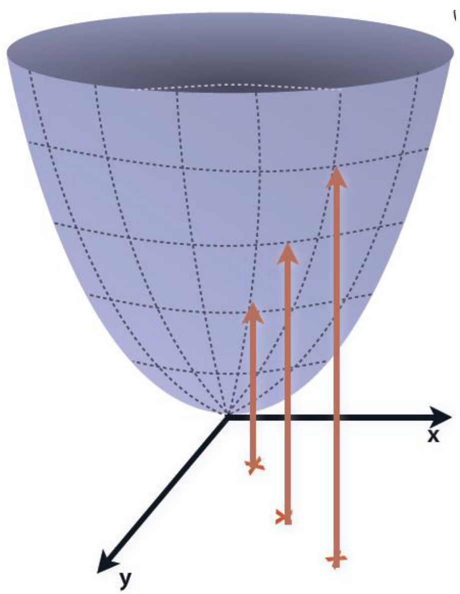
MVD local pattern recognition

`PndMvdRiemannTrackFinderTask`

❖ The *xy* coordinates of the hits are translated to a **Riemann surface**

❖ In particular here the *xy* coordinates are mapped to a circular paraboloid, via the transformation

$$\mathbf{w = x^2 + y^2}$$



❖ The equation of a circle in the *xy* plane

$$\mathbf{(x - x_0)^2 + (y - y_0)^2 = R_0^2}$$

in the *xyw* space becomes

$$\mathbf{w - 2\,x_0 \cdot x - 2\,y_0 \cdot y + x_0^2 + y_0^2 - R_0^2 = 0}$$

❖ A plane is fitted to the mapped points and the parameters of the plane are then transformed back to the circle parameters in the *xy* plane.

❖ In a second stage a linear fit with a straight line is made between the arclength of the points on the fitted circle *vs* the z coordinate of the hits

[R. Fruhwirth et al. *Helix fitting by an extended Riemann fit* NIM A 490(1-2):366-378, 2002.]

STT local pattern recognition

`PndSttFindTracks/PndSttRealTrackFinder`

❖ **constraint:** the track is forced to come from (0, 0, 0), so it applies only to **primary tracks**

❖ The hits in the STT are ordered from outside to inside

❖ they are transformed through a conformal map into a conformal space where tracks which are circles in *xy* plane become straight lines

$$\mathbf{u} = \frac{\mathbf{x}}{\mathbf{x}^2 + \mathbf{y}^2}, \quad \mathbf{v} = \frac{\mathbf{y}}{\mathbf{x}^2 + \mathbf{y}^2}$$

❖ the hits are clusterized and each cluster is fitted in the conformal plane. The parameters of the fitted line are transformed back to the circle parameters in the *xy* plane

❖ an iterative procedure is used to add more hits, with a distance criterion

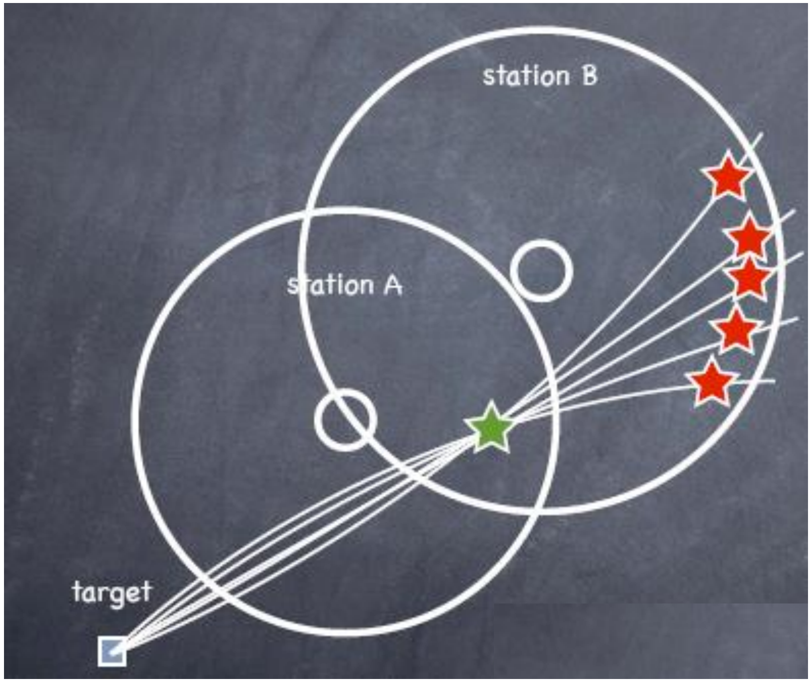
WARNING! *The track finding for the* ***secondary tracks*** *is still under construction and so will not be explained here*

GEM local pattern recognition

`PndGemFindTracks/PndGemTrackFinderOnHits`

❖ Hits on the same station are matched:
each station contains two sensors, The hits of one sensor are matched to the ones of the other one, following a distance criterion

❖ Hits from different stations are matched to get tracklet:
tracks coming from the IP are extrapolated from one station to another trying to match the nearest hits on it



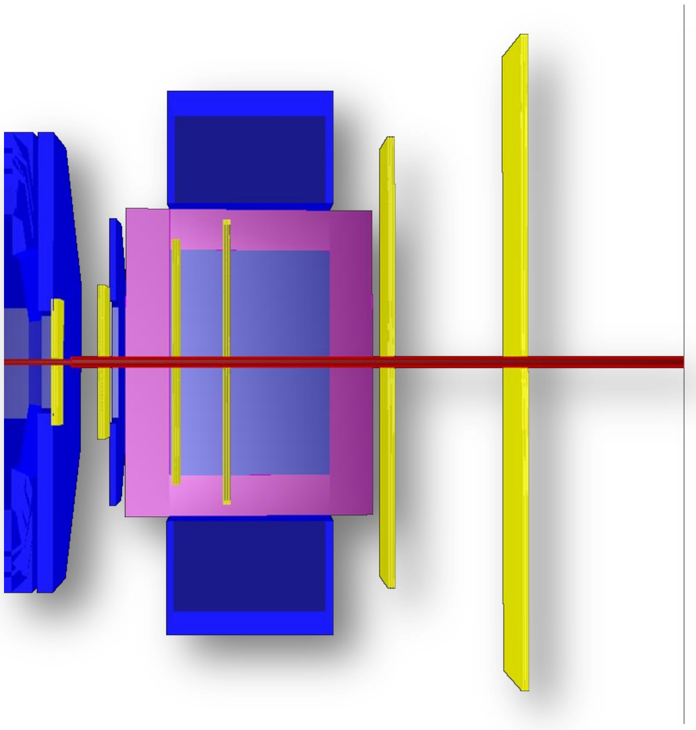
❖ The tracklets are connected to get tracks

❖ Cleanup and creation of the PndTrackCand

FTS local pattern recognition

`PndFtsTrackerIdeal`

- It uses MVD + GEM + FTS hits for the very forward boosted tracks
- Only an ideal pattern recognition exists here
- It provides the possibility to:
  - smear the vertex position
  - smear the momentum
  - set a tracking efficiency



Global pattern recognition

`PndSttMvdTrackFinder/PndSttMvdGemTrackFinder`

❖ this is the default global PR, the one we used in the CT TDR.

❖It contains: MVD + STT + GEM

❖ It starts from the MVD and STT standalone track finders

❖ It extrapolates the STT alone found helix to the MVD to match its hits

❖ It extrapolates the MVD alone found helix to the STT to match its hits

❖ The new track candidates, of both STT and MVD hits, are refitted with a helix, to improve the parameters thanks to the high resolution of the MVD

❖ Once the MVD + STT track is ready, it is extrapolated to the GEM station to collect also the GEM hits

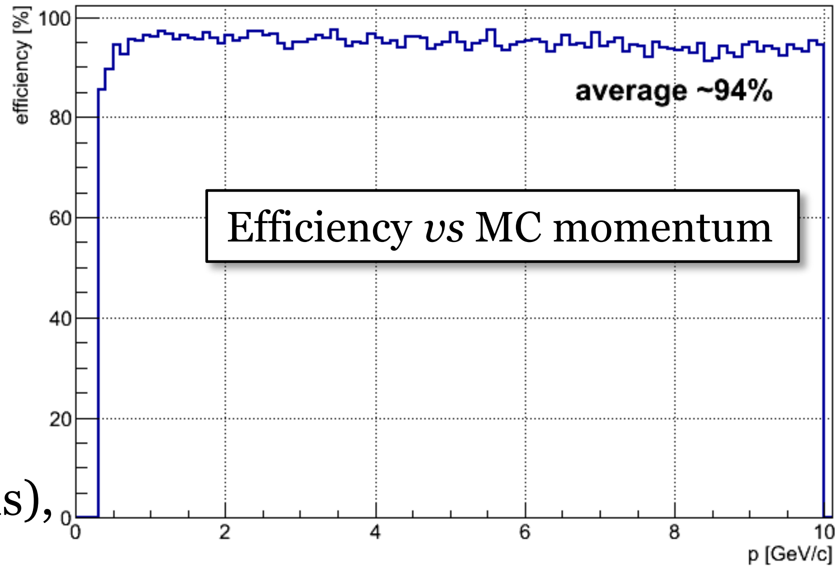
❖ At this stage no refit is done since the GEM stations are in a zone where the field is non homogeneous and so the helix is not suitable for fitting.

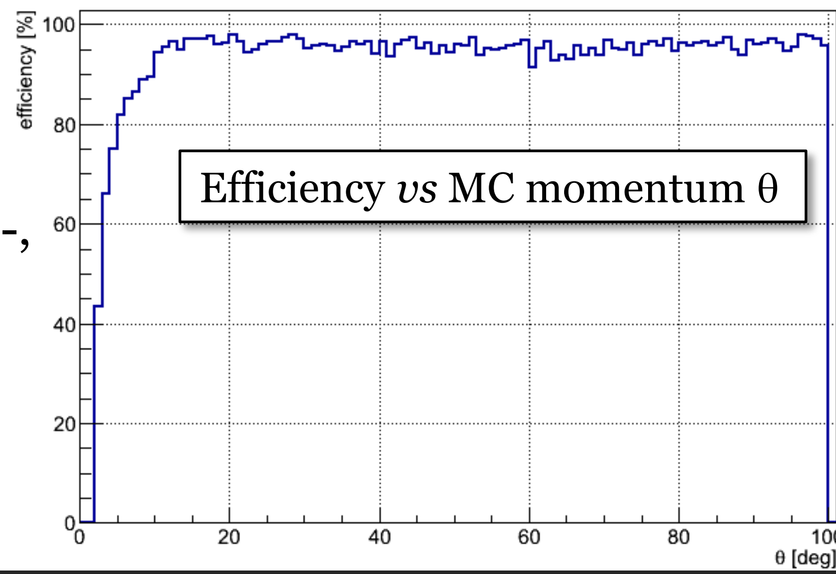
Global pattern recognition (2)

`PndBarrelTrackFinder`

- this is an alternate global PR

- **scheme**
  - It does not start in any specific detector
  - It treats MVD, STT, GEM hits the same way
  - It looks for correlations in *xy* plane
  - Tries to gather *z* information when possible
  - Hit mixing (to remove unphysical correlations), reduce "for" loop nesting (to increase speed)





- **results**
  - 94% average tracking efficiency (basing on 10000 events with 4 muons: 2μ+, 2μ-, 0.3< p< 10 GeV/c)
  - almost no dependence on θ (drop below 10°, no ~20° problem), almost no dependence on momentum

Pattern Recognition Summary

❖ The pattern recognition algorithms take as **input** the TClonesArray of PndXXXHit

❖ They group the hits following various criteria and algorithms in lists, stored in PndTrackCand objects

❖ They fit these lists to obtain a preliminary guess of the track parameters, storing it in PndTrack objects

❖ They register the track and track candidates to **output**

YOUR TURN!

EXERCISE 1

try to write a task for the ideal pattern recognition of the TOY detector

Hints:

INPUT: PndToyHit, PndToyPoint *...to obtain...*
OUTPUT: PndTrackCand

INPUT: PndMCTrack *...to obtain...*
OUTPUT: PndTrack

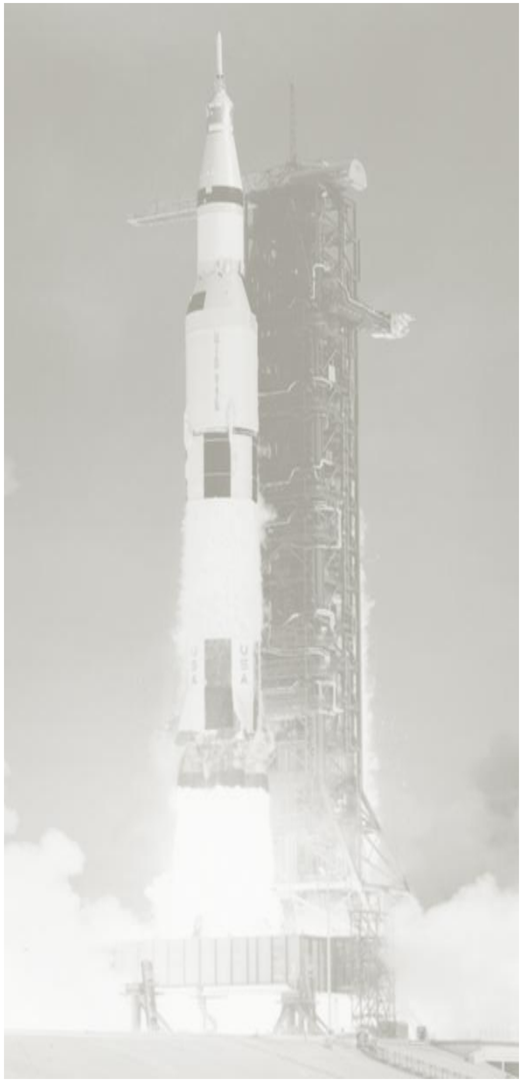
Summary

- ❖ Introduction
- ❖ Input/Output Data
- ❖ Pattern Recognition
- ❖ Track Fitting
- ❖ Track Extrapolation
- ❖ Some macros…

The Kalman filter

❖ R. E. Kalman, MIT engineer, proposed this method in 1961 in the framework of the control and optimization theory of systems

❖ Fruehwirth, in 1987, applied the method as a useful track fitting technique in particle physics



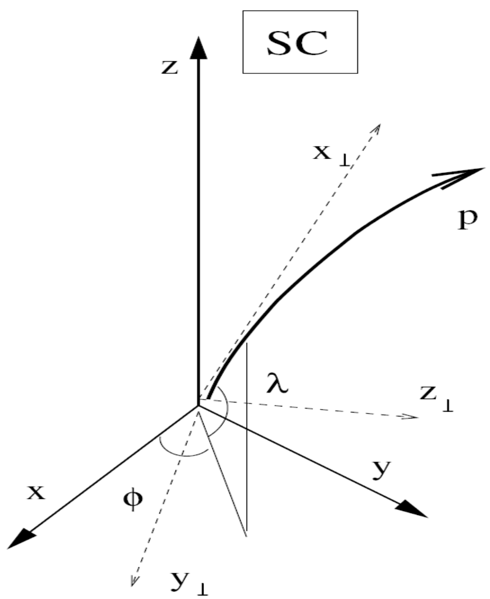
Kalman filters were applied to the navigation system during the Apollo program and on the Space Shuttle, moreover they were used for submarines, unmanned aerospace vehicles and weapons (*e.g.* cruise missiles).

Track representation

❖ The physical path of a particle of assigned mass *m* and momentum *p* is a six-fold entity of parameters *x, y, z, px, py, pz*

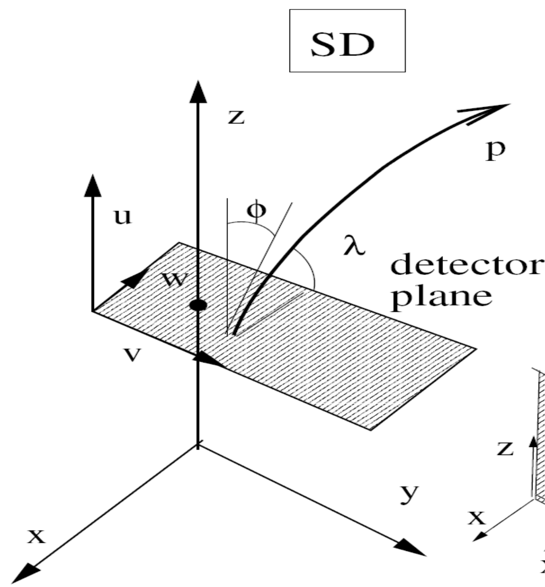
❖ The track is defined as a set of points in the detectors, corresponding to the intersections of the physical path of a particle with the detector planes
→ among the six parameters, one is fixed by the measurement

curvilinear frame



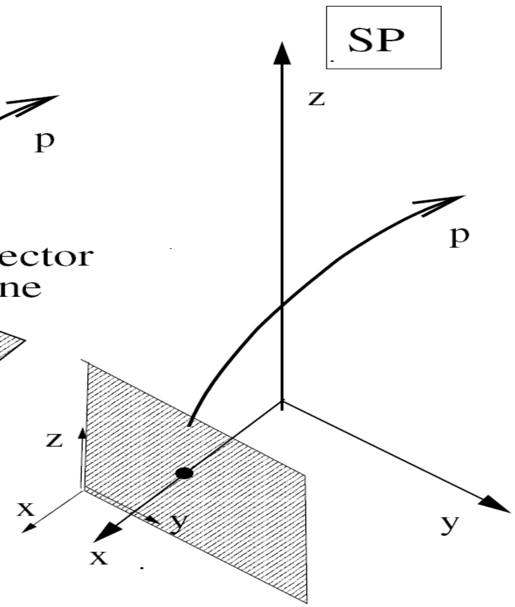
$$\frac{1}{p}, \lambda, \varphi, y_{\perp}, z_{\perp}$$

detector frame



$$\frac{1}{p}, v', w', v, w$$

plane frame



$$\frac{1}{p}, y', z', y, z$$

The Kalman filter

❖ Without entering in too many details, it is based on the minimization of the following χ^2, in 5-dimensional space

$$\chi^2(\mathbf{f}) = (\mathbf{x}-\mathbf{f})\mathbf{V}(\mathbf{x}-\mathbf{f}) + (\mathbf{e}-\mathbf{f})\mathbf{W}(\mathbf{e}-\mathbf{f})$$

f	is the parameter vector on each plane
x	is the measurement vector
V	is the weight matrix connected to the measurement
e	is the estimated parameter vector on i^{th} plane given the one on plane $(i-1)^{th}$
W	is the weight matrix connected to the extrapolation

❖ It is a local recursive fitting method which takes into account possible inhomogeinity of the magnetic field and the effect of the materials.

❖ It combines prediction and observation on each measurement plane in order to correct the trajectory estimation

Just to give an idea!

The Kalman filter

When applied to N hits in 5 dimensional space, the minimization of the χ^2 can be performed in three steps:

PREDICTION *the track parameters on the plane i are inferred starting from the knowledge gained up to plane i – 1* → $\boldsymbol{e}_i$

$$\boldsymbol{e}_i \equiv \boldsymbol{e}_i(\boldsymbol{k}_{i-1}) = \boldsymbol{G}(\boldsymbol{k}_{i-1})$$

$$\boldsymbol{\sigma}^2[\boldsymbol{e}_i] = \boldsymbol{T}(l_i, l_{i-1})\,\boldsymbol{\sigma}^2[\boldsymbol{k}_{i-1}]\,\boldsymbol{T}^T(l_i, l_{i-1}) + \boldsymbol{W}_{i-1,i}^{-1}$$

FILTERING *a preliminary value of the track parameter* $\boldsymbol{k}_i$ *is evaluated as a "weighted mean" between the measured* $\boldsymbol{x}_i$ *and the predicted value* $\boldsymbol{e}_i$

$$\boldsymbol{k}_i = \boldsymbol{\sigma}^2[\boldsymbol{k}_i]\left(\boldsymbol{\sigma}^{-2}[\boldsymbol{e}_i]\,\boldsymbol{e}_i + \boldsymbol{V}_i\boldsymbol{x}_i\right)$$

$$\boldsymbol{\sigma}^{-2}[\boldsymbol{k}_i] = \boldsymbol{\sigma}^{-2}[\boldsymbol{e}_i] + \boldsymbol{V}_i$$

SMOOTHING *the final estimate of the parameters* $\boldsymbol{f}_i$ *is calculated*

$$\boldsymbol{f}_i = \boldsymbol{k}_i + \boldsymbol{A}_i\left(\boldsymbol{f}_{i+1} - \boldsymbol{e}_{i+1}\right)$$

$$\boldsymbol{\sigma}^2[\boldsymbol{f}_i] = \boldsymbol{\sigma}^2[\boldsymbol{k}_i] + \boldsymbol{A}_i\left(\boldsymbol{\sigma}^2[f_{i+1}] - \boldsymbol{\sigma}^2[\boldsymbol{e}_{i+1}]\right)\boldsymbol{A}_i^T$$

$$\boldsymbol{A}_i = \boldsymbol{\sigma}^2[\boldsymbol{k}_i]\,\boldsymbol{T}^T(l_{i+1}, l_i)\,\boldsymbol{\sigma}^{-2}[\boldsymbol{e}_{i+1}]$$

Just to give an idea!

The Kalman filter

Forget the math, just get the idea!

PREDICTION

Predict the track parameters on next plane, knowing them at the previous one

FILTERING

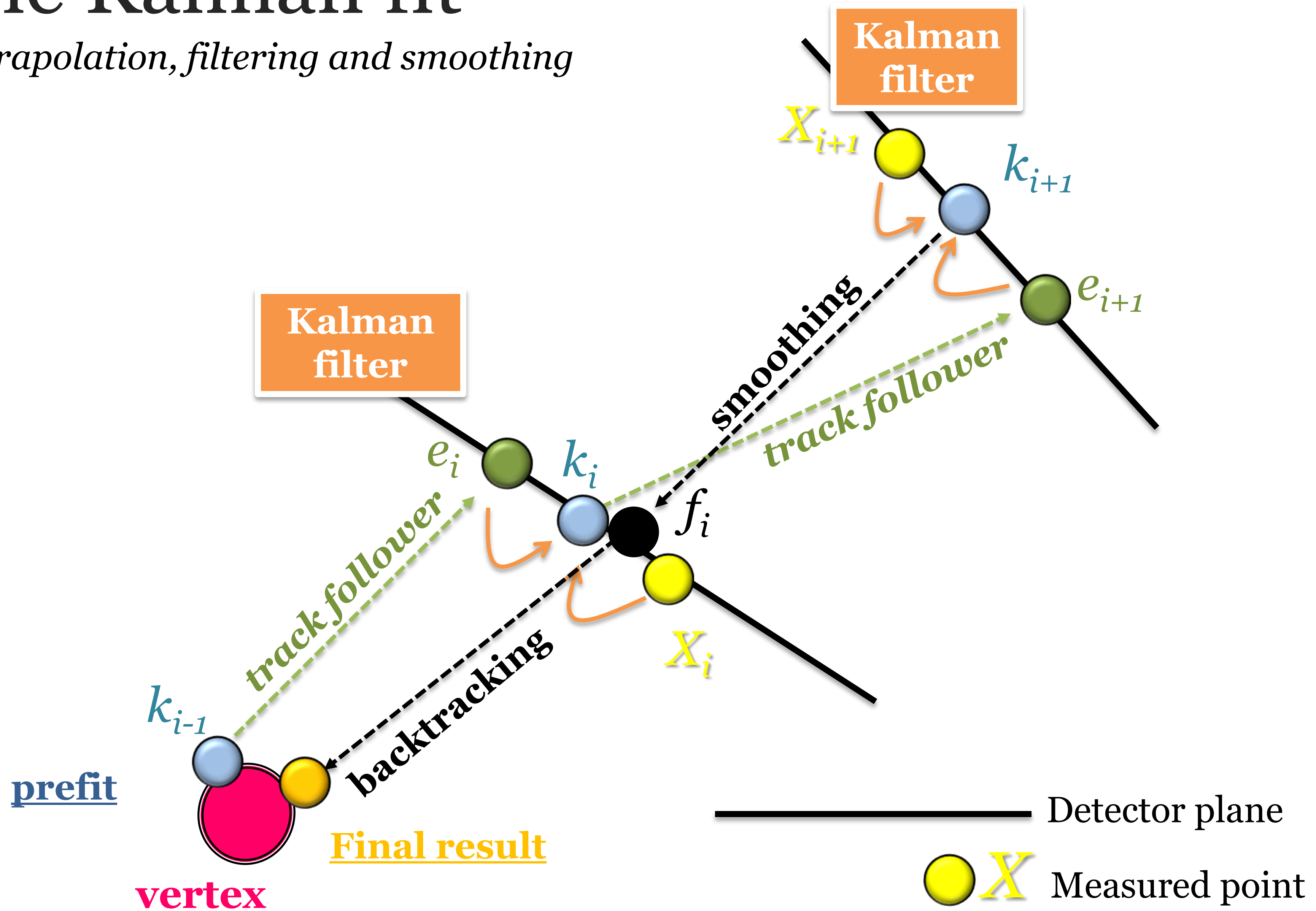
Make a weighted mean between your prediction and the measurement

SMOOTHING

...or substitute it with a back-Kalman

The Kalman fit

Extrapolation, filtering and smoothing

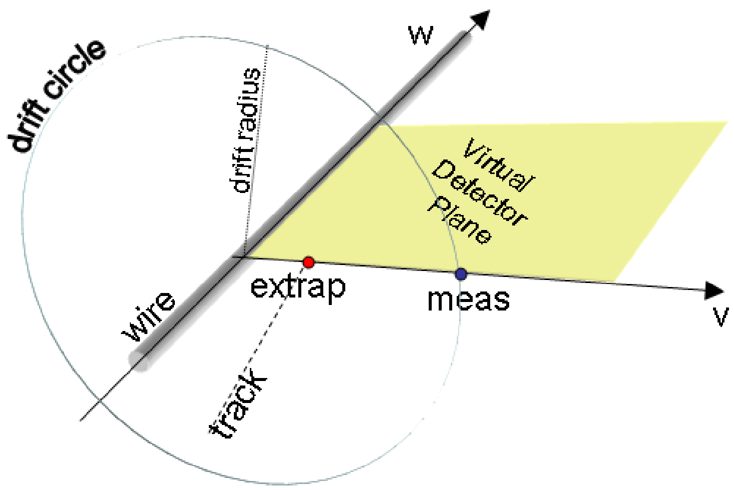


Detector planes

❖ In pandaroot we always perform the filtering step on a plane

❖ Planes can be *real* (e.g. a Silicon Detector plane) or *virtual* (e.g. built at extrapolation time, used for non planar devices, such as the STT)

❖ They are defined by the origin and the unit vectors spanning the plane



The Kalman fit in pandaroot

❖ the two *leading actors* of PANDA Kalman fit are:

- ❖ **GENFIT**
- ❖ **GEANE**

❖ **GENFIT** is a tool which is able to handle the information coming from different trackers, merge them and use all of them in the **Kalman** fitting procedure, giving the fitted track as output

❖ **GEANE** is the **track follower**, which is used in order to propagate the parameters mean values and the covariance matrices from one plane to another in the detector.

GENFIT

GENERIC FIT,
TOOLKIT FOR TRACK RECO

It matches the info from the different subdetectors to be able to Kalman fit them together

---

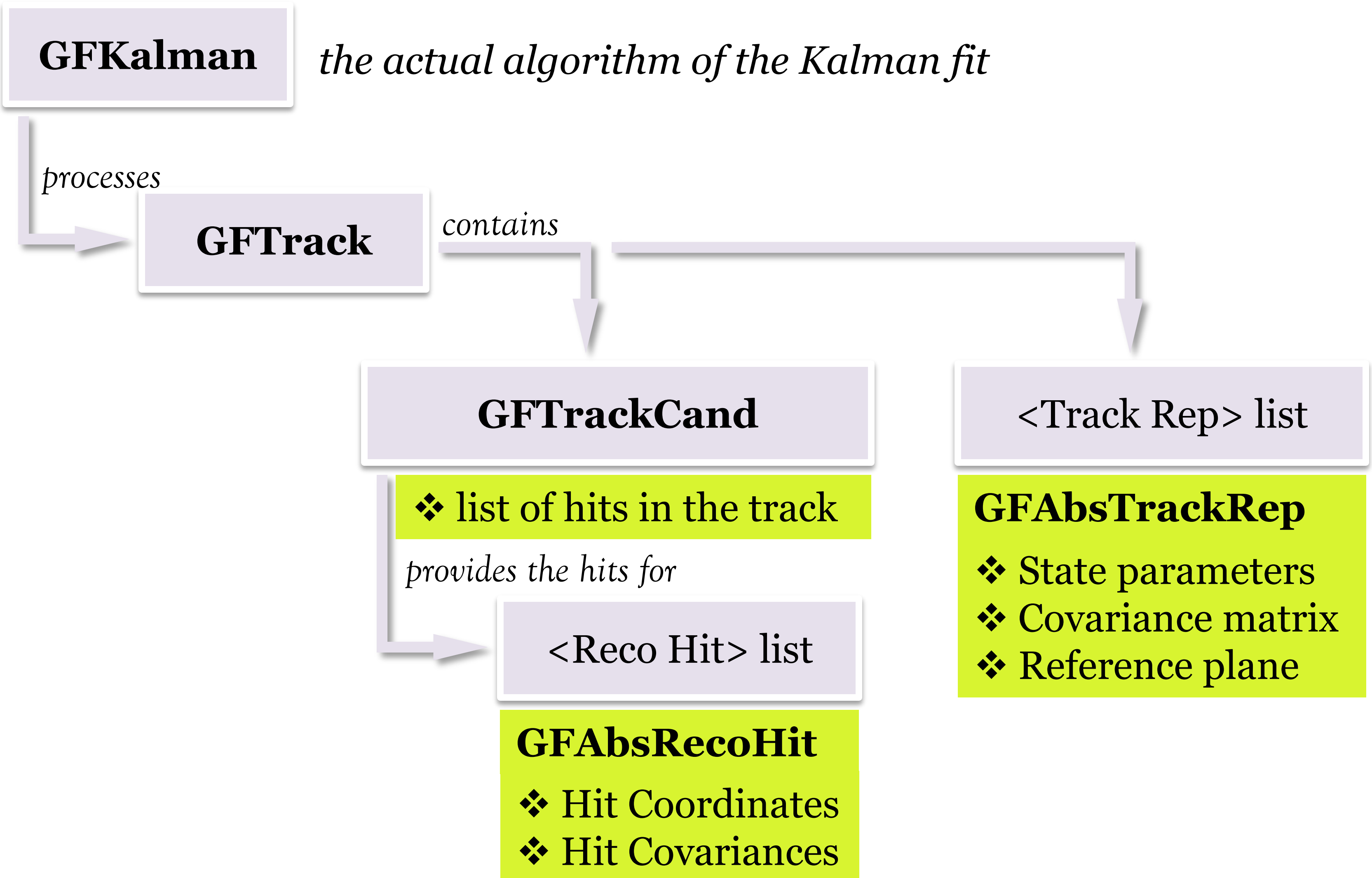
❖ it was developed by the TUM Munich group

❖ it is downloaded from the external repository in *sourceforge*:
http://genfit.sourceforge.net

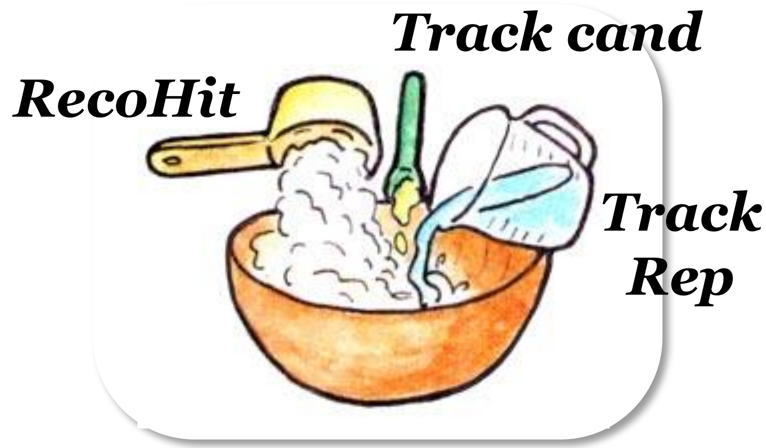
❖ once you download pandaroot, the external link to the original repository is read by svn and you will get it automatically

❖ you will find it in the directory: `genfit`

GENFIT structure



GENFIT structure



GFKalman

GFTrack

sorted, from PR — **GFTrackCand**

- ❖ list of hits in the track

USER — <Reco Hit> list

GFAbsRecoHit

- ❖ Hit Coordinats
- ❖ Hit Covariances

USER — <Track Rep> list

GFAbsTrackRep

- ❖ State parameters
- ❖ Covariance matrix
- ❖ Reference plane

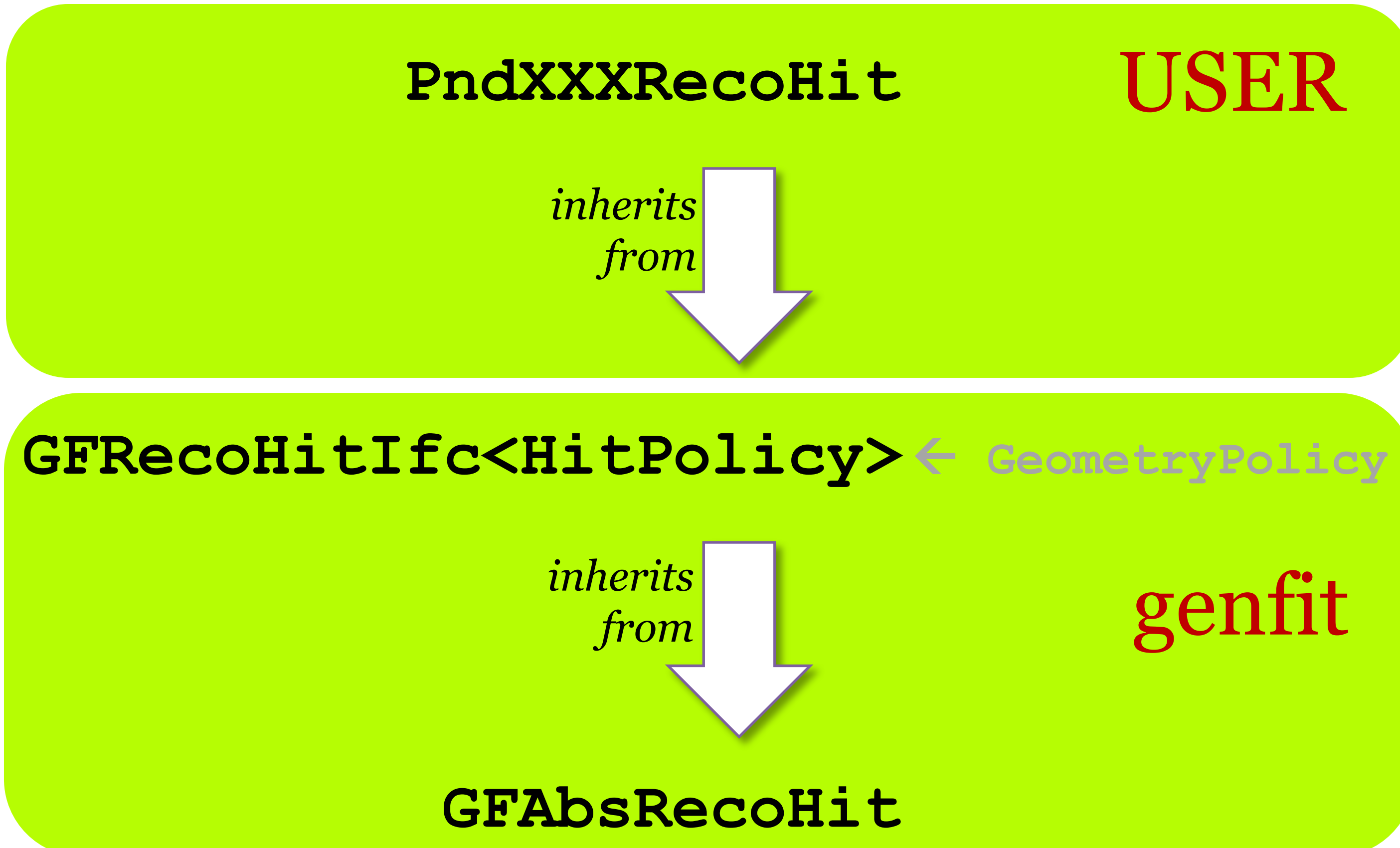
Ingredient 1: the RecoHit

❖ the RecoHit, PndXXXRecoHit, is the class which contains the “translation” of the measured quantities from the PndXXXHit to the variables that GENFIT can handle in order to use them in the fit along with the other detectors

❖ the RecoHit contains:
- ❖ hit coordinates
- ❖ covariance matrix
- ❖ detector plane description
- ❖ measurement matrix H

❖ you will find the existing RecoHits in `GenfitTools/recohits`

The detector RecoHit



GFAbsRecoHit

This is the abstract reco hit

- **class members**
  - hit coordinates
  - covariance matrix
  - number of measured variables

```
TMatrixT<double> fHitCoord;
TMatrixT<double> fHitCov;
int fNparHit;
```

- **virtual functions**

Implemented in the HitPolicy class

  - Access the hit coordinates on the detector plane

```
virtual TMatrixT<double> getHitCoord(const GFDetPlane&)
```

  - Access the covariance matrix on the detector plane

```
virtual TMatrixT<double> getHitCov(const GFDetPlane&)
```

  - Retrieve the detector plane

```
virtual const GFDetPlane& getDetPlane(GFAbsTrackRep*)
```

  - Write the H, measurement matrix, to perform the transformation from the measurement frame to the parameters frame

```
virtual TMatrixT<double> getHMatrix(const GFAbsTrackRep* stateVector)
```

GFRecoHitIfc

This is the "intermediate " reco hit

- It is a template class, which allows to use different policy classes:
  - GFRecoHitIfc<PlanarHitPolicy> a basic planar hit (MVD, GEM)
  - GFRecoHitIfc<SpacepointHitPolicy> a basic space point hit (MDT)
  - GFRecoHitIfc<WireHitPolicy> a basic hit on a wire (FTS, STT)

- These policy classes are available in `genfit`

- Each policy implements the virtual functions of the mother class GFAbsRecoHit:

```
virtual TMatrixT<double> getHitCoord(const GFDetPlane&)
```

```
virtual TMatrixT<double> getHitCov(const GFDetPlane&)
```

```
virtual const GFDetPlane& getDetPlane(GFAbsTrackRep*)
```

You have to choose the most suitable hit policy depending on the kind of detector hit you get. Basically on its geometry

RecoHit Summary

❖ To insert a detector in GENFIT, you need to implement your own PndXXXRecoHit

❖ It inherits from GFRecoIfc<HitPolicy>, so you have to choose your **policy**.
To do so, look at your hit geometry: is it a spatial hit, a planar hit, just a drift radius?
Choose the policy accordingly (they are in `genfit`).
You can also write your own one (usually not needed! So think twice ;-))

❖ Provide as **input** to the PndXXXRecoHit your PndXXXHit:
the RecoHits are created automatically by the hit producer, so this is the right input to give. You can also give extra inputs, but you might need to implement your own hit producer…

❖ Write the **measurement matrix H** to connect the measurement and the parameters space

…and you are done!

YOUR TURN!

EXERCISE 2

try to write your own PndToyRecoHit for the TOY detector

Hints:

INPUT: PndToyHit
what sort of detector response do you get from a silicon detector? → *policy*

The RecoHitFactory

❖ The GFRecoHitFactory is the class which takes care of *transforming* the PndXXXHit into PndXXXRecoHit

```
GFRecoHitFactory *rhf = new GFRecoHitFactory();
```

❖ The class needs the TClonesArray where to retrieve the hits, together with the corresponding detector Id.
This is done with the GFRecoHitProducer:

```
rhf->addProducer(detectorID,
     new GFRecoHitProducer<PndXXXHit,PndXXXRecoHit>(hitarray))
```

REMEMBER? The RecoHit ctor `PndXXXRecoHit(PndXXXHit *hit)`

GFRecoHitProducer

❖ It is a **template** class <class hit_T, class recoHit_T>

```
GFRecoHitProducer<class hit_T,class recoHit_T>
```

❖ **ctor**

It takes in input the TClonesArray of the hits

```
GFRecoHitProducer(TClonesArray*);
```

❖ Is associates the PndXXXHit (hit_T) to the correct PndXXXRecoHit (recoHit_T) to build the RecoHit out of the index-th

```
template <class hit_T,class recoHit_T> GFAbsRecoHit*
GFRecoHitProducer<hit_T,recoHit_T>::produce(int index) {
…
return ( new recoHit_T( (hit_T*) hitArrayTClones->At(index)));
}
```

Ingredient 2: Track Cand

❖ It has the same basic idea of the PndTrackCand

❖ It contains two std::vector<int>:
  - ❖ list of the **detector Ids**
  - ❖ list of the **hit Ids**, i.e. the indices in the corresponding TClonesArray

❖ The list of hits must come from the **pattern recognition**

❖ It must be already **sorted**

❖ It will be passed to the GFTrack (analogy with PndTrackCand ←→ to PndTrack)

```
GFTrack *track = new GFTrack(trackrep)
track-setCandidate(trackcand);
```

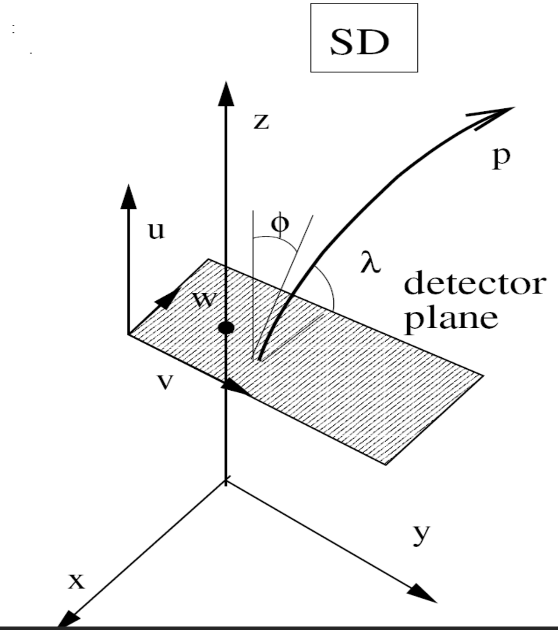
❖ And it will provide the hits to the RecoHitFactory

```
track->addHitVector(rhf->createMany(trackcand);
```

Ingredient 3: Track Rep

- the track representation contains:
  - the state 5 components vector
  - the covariance 5 X 5 matrix
  - the reference plane

- the measurement matrix H is used to transform the state vector from the frame of the parameters to the measurement one, e.g.:



$$\mathit{meas} = \mathbf{H} \cdot \mathrm{state}$$

$$\begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{1/p} \\ \mathbf{v'} \\ \mathbf{w'} \\ \mathbf{v} \\ \mathbf{w} \end{pmatrix}$$

GFAbsTrackRep

This is the abstract track rep

❖ it contains the functions to call the external track follower to perform the extrapolation step in the Kalman fit

❖ different extrapolations are available (*see later the track follower description*)

```
virtual double extrapolate(const GFDetPlane& plane,
                             TMatrixT<double>& statePred,
                             TMatrixT<double>& covPred)

virtual void extrapolateToPoint(const TVector3& point,
                                 TVector3& poca,
                                 TVector3& normVec)

virtual void extrapolateToLine(const TVector3& point1,
                                const TVector3& point2,
                                TVector3& poca,
                                TVector3& normVec,
                                TVector3& poca_onwire)
```

GeaneTrackRep

❖It is one of the GFAbsTrackRep possible implementations, the one we ususally use in pandaroot

❖It “lives” in `GenfitTools/trackrep/GeaneTrackRep`

❖ **ctor**

```
GeaneTrackRep(FairGeanePro* geane,
          const GFDetPlane& plane,
          const TVector3& mom,
          const TVector3& poserr,
          const TVector3& momerr,
          int q,
          int PDGCode)
```

❖ It contains the actual implemetation of the extrapolation functions by means of the external track follower GEANE

PndGenfitAdapters

- To transform the PndTrack(Cand) into GFTrack(Cand) use the adapters inside `PndGenfitAdapters`

- it "lives" in `GenfitTools/adapters/`

- **functions**
  - GFTrackCand → PndTrackCand

```
PndTrackCand* GenfitTrackCand2PndTrackCand(const GFTrackCand* cand)
```

  - PndTrackCand → GFTrackCand

```
GFTrackCand* PndTrackCand2GenfitTrackCand(PndTrackCand* cand)
```

  - GFTrack → PndTrack

```
PndTrack* GenfitTrack2PndTrack(const GFTrack* trk)
```

Kalman fit Summary (1)

To insert your detector in GENFIT, you need to:

❖ implement your own **reco hit and create a** RecoHitFactory.
Tell the RecoHitFactory which is the TClonesArray of your detector

```
GFRecoHitFactory *rhf = new GFRecoHitFactory();
rhf->addProducer(detectorID,
 new GFRecoHitProducer<PndXXXHit,PndXXXRecoHit>(hitarray))
```

❖ create your **track representation** with the initial values, e.g. extrapolating back the pattern recognition found track to the vertex and

```
GeaneTrackRep *trackrep = new GeaneTrackRep(geanePro,
                          StartPlane, StartMom,
                          StartPosErr, StartMomErr,
                          charge, PDGCode);
```

Track follower

from PR

Kalman fit Summary (2)

To insert your detector in GENFIT, you need to:

- create your **GFTrackCand**

```
GFTrackCand *trackcand = transform PndTrackCand into GFTrackCand
```

- create your **GFTrack** and feed it with the track representation and the track cand

```
GFTrack *track = new GFTrack(trackrep)
track->setCandidate(trackcand);
```

- add the hit vector you get from the reco hit factory after applying it to the track cand

```
std::vector<*GFAbsRecoHit> hitlist =
                     rhf->createMany(trackcand);
track->addHitVector(hitlist);
```

- process the track

```
GFKalman genfitter;
genfitter.processTrack(trk);
```

- translate back to PndTrack

Summary

- ❖ Introduction
- ❖ Input/Output Data
- ❖ Pattern Recognition
- ❖ Track Fitting
- ❖ Track Extrapolation
- ❖ Some macros…

GEANE

TRACK FOLLOWER:
PROPAGATES THE TRACK PARAMETERS AND THEIR ERRORS

Simplified ***geant3*** *tracking algorithms*

\+

Error propagation *routines from EMC*

❖ the original code is in FORTRAN

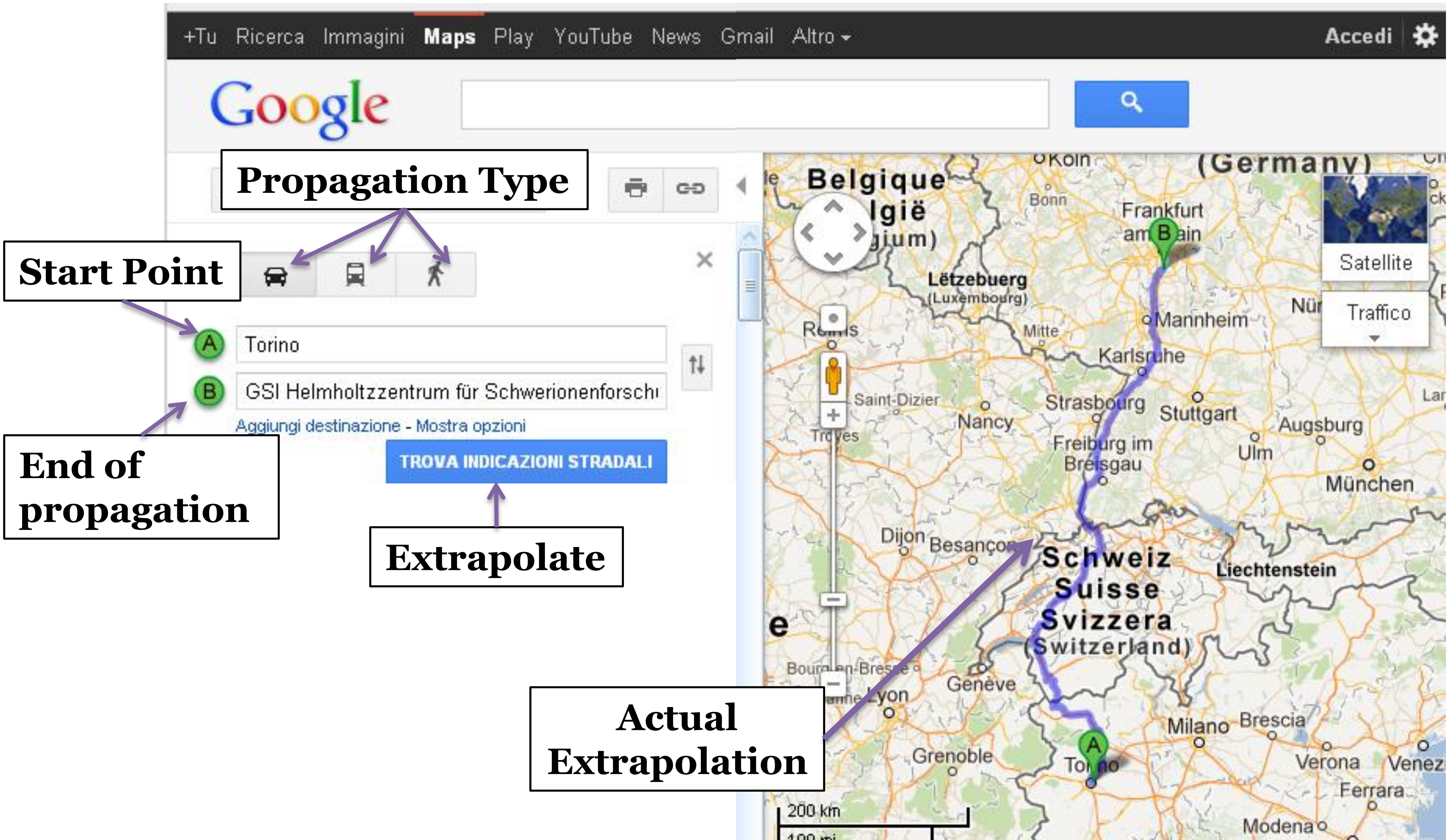
❖ it is distributed with the geant3 by CERN

❖ now, an interface with the VMC has been developed and you can find it fairroot

❖ in your downloaded pandaroot, you will find the interface in the packages `geane` and `trackbase`

GEANE

...It is like Google Maps, if you will ;-)



Just to give an idea!

The Kalman filter

When applied to N hits in 5 dimensional space, the minimization of the χ^2 can be performed in three steps

PREDICTION *the track parameters on the plane i are inferred starting from the knowledge gained up to plane i – 1* → $\boldsymbol{e}_i$

$$\boldsymbol{e}_i \equiv \boldsymbol{e}_i(\boldsymbol{k}_{i-1}) = \boldsymbol{G}(\boldsymbol{k}_{i-1})$$

$$\boldsymbol{\sigma}^2[\boldsymbol{e}_i] = \boldsymbol{T}(l_i, l_{i-1})\,\boldsymbol{\sigma}^2[\boldsymbol{k}_{i-1}]\,\boldsymbol{T}^T(l_i, l_{i-1}) + \boldsymbol{W}^{-1}_{i-1,i}$$

FILTERING *a preliminary value of the track parameter* $\boldsymbol{k}_i$ *is evaluated as a "weighted mean" between the measured* $\boldsymbol{x}_i$ *and the predicted value* $\boldsymbol{e}_i$

$$\boldsymbol{k}_i = \boldsymbol{\sigma}^2[\boldsymbol{k}_i]\left(\boldsymbol{\sigma}^{-2}[\boldsymbol{e}_i]\,\boldsymbol{e}_i + \boldsymbol{V}_i \boldsymbol{x}_i\right)$$

$$\boldsymbol{\sigma}^{-2}[\boldsymbol{k}_i] = \boldsymbol{\sigma}^{-2}[\boldsymbol{e}_i] + \boldsymbol{V}_i$$

SMOOTHING *the final estimate of the parameters* $\boldsymbol{f}_i$ *is calculated*

$$\boldsymbol{f}_i = \boldsymbol{k}_i + \boldsymbol{A}_i\left(\boldsymbol{f}_{i+1} - \boldsymbol{e}_{i+1}\right)$$

$$\boldsymbol{\sigma}^2[\boldsymbol{f}_i] = \boldsymbol{\sigma}^2[\boldsymbol{k}_i] + \boldsymbol{A}_i\left(\boldsymbol{\sigma}^2[f_{i+1}] - \boldsymbol{\sigma}^2[\boldsymbol{e}_{i+1}]\right)\boldsymbol{A}_i^T$$

$$\boldsymbol{A}_i = \boldsymbol{\sigma}^2[\boldsymbol{k}_i]\,\boldsymbol{T}^T(l_{i+1}, l_i)\,\boldsymbol{\sigma}^{-2}[\boldsymbol{e}_{i+1}]$$

The track follower

❖ In the **prediction step** of the Kalman filter the state at plane i is inferred starting from the knowledge of the state at plane i-1

$$\boldsymbol{e}_i \equiv \boldsymbol{e}_i(\boldsymbol{k}_{i-1}) = \boldsymbol{G}(\boldsymbol{k}_{i-1})$$

$$\boldsymbol{\sigma}^2[\boldsymbol{e}_i] = \boldsymbol{T}(l_i, l_{i-1})\,\boldsymbol{\sigma}^2[\boldsymbol{k}_{i-1}]\,\boldsymbol{T}^T(l_i, l_{i-1}) + \boldsymbol{W}_{i-1,i}^{-1}$$

❖ both the **mean values of the parameters** (5) and their **covariance matrix** (5 X 5) are necessary at filtering step

$$\boldsymbol{k}_i = \boldsymbol{\sigma}^2[\boldsymbol{k}_i]\left(\boldsymbol{\sigma}^{-2}[\boldsymbol{e}_i]\,\boldsymbol{e}_i + \boldsymbol{V}_i\boldsymbol{x}_i\right)$$

$$\boldsymbol{\sigma}^{-2}[\boldsymbol{k}_i] = \boldsymbol{\sigma}^{-2}[\boldsymbol{e}_i] + \boldsymbol{V}_i$$

GEANE intro

- ❖ GEANE extrapolates both the **mean values** of the parameters & their **covariance matrix**

- ❖ It does this taking using the same MC banks for the geometry

- ❖ It takes into account:
  - ❖ material effects
  - ❖ magnetic field
  - ❖ physical effects

- ❖ it uses different reference frames, with the possibility to change among them

- ❖ different kind of propagation are available

Error transportation (1)

❖ the error matrix propagation is transported this way:

$$\sigma^2(x) = T(x, x_o)\,\sigma^2(x_o)\,T^T(x, x_o) + R^{-1}(x)$$

Where $\boldsymbol{\sigma^2(x)}$ is the covariance matrix @ $\mathbf{x}$
$\mathbf{T(x, x_o)}$ is the transport matrix from $\mathbf{x_o}$ to $\mathbf{x}$
$\mathbf{R^{-1}(x)}$ is the random noise contribution

❖ the **transport matrix** contains different contributions:

$$T = I + \left(A_{x+dx} + B_{x+dx}\right) \cdot dx$$

Where:
❖ **I** is the identity, to propagate the initial error as it is and add the other contributions
❖ $\mathbf{A_{x,\, x+dx}}$ is the contribution of the error on the direction (also in absence of magnetic field)
❖ $\mathbf{B_{x,\, x+dx}}$ is the contribution due to the magnetic field

Error transportation (2)

Just to give an idea!

$$A = \begin{pmatrix} \dfrac{\left(\dfrac{\partial^2 \, {}^{1}\!/_{p}}{\partial l^2}\right)}{\left(\dfrac{\partial \, {}^{1}\!/_{p}}{\partial l}\right)} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \cos\lambda & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ H_2 & 0 & -{}^{H_0}\!/_{p} & {}^{H_2 H_3}\!/_{p^2} & -{}^{H_2^2}\!/_{p^2} \\ -\dfrac{H_3}{\cos\lambda} & \dfrac{H_0}{p\cos^2\lambda} & \dfrac{H_2 \tan\lambda}{p} & -\dfrac{H_3^2}{p^2\cos\lambda} & -\dfrac{H_2 H_3}{p^2\cos\lambda} \\ 0 & 0 & 0 & 0 & -\dfrac{H_3 \tan\lambda}{p} \\ 0 & 0 & 0 & \dfrac{H_3 \tan\lambda}{p} & 0 \end{pmatrix}$$

GEANE interface

In `geane` you will find:

❖ **`FairGeane`**: this is the actual GEANE task. It reads the files which contain all the cuts and the information on which model to use to consider the various physical effects.

WARNING! When you want to use GEANE you need to add this task to the FairRunAna tasks

❖ **`FairGeanePro`**: this contains all the propagation stuff

In `trackbase` you will find:

❖ **`FairTrackPar/ParP/ParH`**: these are the track representation in GEANE. They are analogous to the GeaneTrackRep in GENFIT

❖ **`FairGeaneUtil`**: this contains the translation to C++ of the FORTRAN routines to perform the frame transformations from/to MARS, SC, SD, SP

In addition to this there is the FairGeaneApplication, which access GEANE @ each step

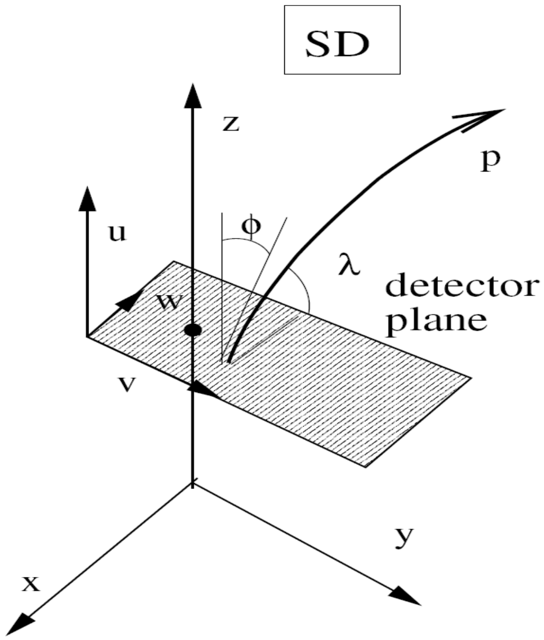
FairTrackPar

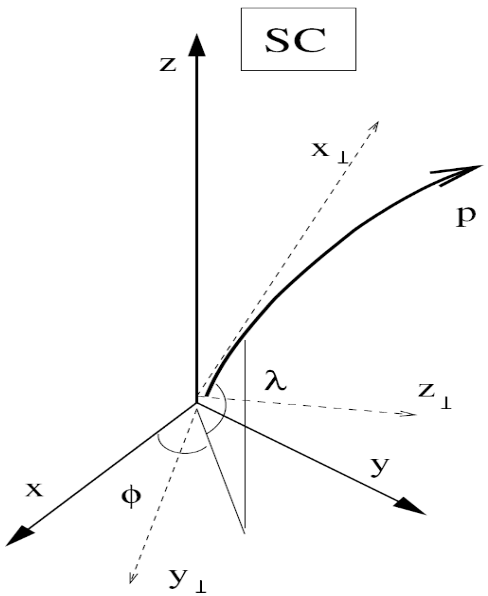
The track parameters base class is FairTrackPar

ctor

```
FairTrackPar(Double_t x, Double_t y, Double_t z,
        Double_t fx, Double_t fy, Double_t fz,
        Int_t q);
```

It has two daughter classes, one for SD (detector system) and one for SC (curvilinear system) description





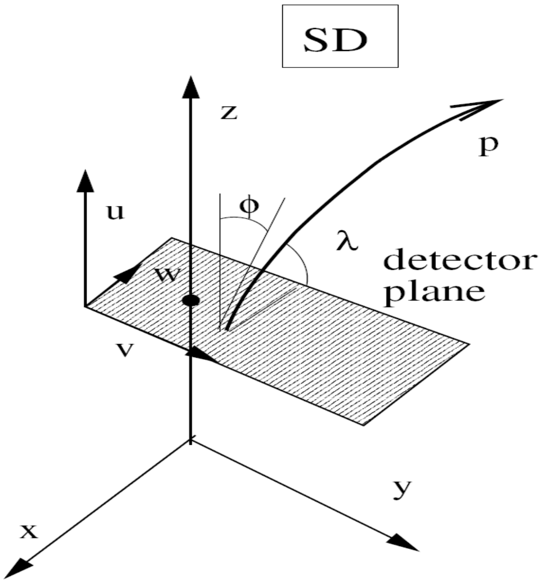
`FairTrackParP` **`FairTrackParH`**

FairTrackParP

❖ Different **ctor**s are available

```
FairTrackParP(Double_t v, Double_t w,
        Double_t Tv, Double_t Tw,
        Double_t qp, Double_t CovMatrix[15],
        TVector3 o, TVector3 dj, TVector3 dk,
        Int_t spu);
```

SD



❖ Diagonal covariance matrix

```
FairTrackParP(TVector3 pos, TVector3 Mom,
        Double_t covMARS[6][6],
        Int_t q,
        TVector3 o, TVector3 dj, TVector3 dk);
```

MARS

❖ Non diagonal covariance matrix

```
FairTrackParP(TVector3 pos, TVector3 Mom,
        TVector3 posErr, TVector3 MomErr,
        Int_t q,
        TVector3 o, TVector3 dj, TVector3 dk);
```

MARS

1/p, v', w', v, w

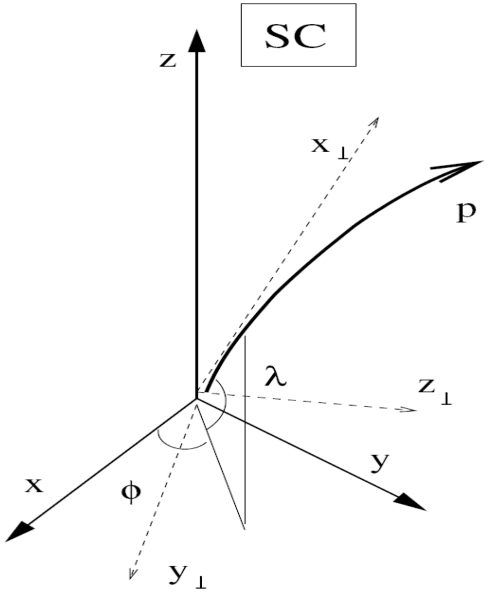
FairTrackParH

❖ Different **ctor**s are available

```
FairTrackParH(Double_t x, Double_t y, Double_t z,
       Double_t lambda, Double_t phi,
       Double_t qp,
MARS   Double_t CovMatrix[15]);
```

```
FairTrackParH(TVector3 pos, TVector3 Mom,
        TVector3 posErr, TVector3 MomErr,
MARS    Int_t q);
```

$1/p, \lambda, \phi, y_\perp, z_\perp$



FairGeanePro

This class contains the propagations functions. Here I put the most used.

❖ Decide which kind of propagation you want:

```
Bool_t PropagateToPlane(TVector3& v0, TVector3& v1, TVector3& v2);
Bool_t PropagateFromPlane(TVector3& v1, TVector3& v2);
Bool_t PropagateToVolume(TString VolName, Int_t CopyNo ,Int_t option);
Bool_t PropagateToLength(Float_t length);
Bool_t PropagateToPCA(Int_t pca, Int_t dir);
```

WARNING! Depending on the kind of propagation your representation must be a FairTrackParP or ParH

❖ Decide which kind of representation you are using as starting and ending ones
WARNING! Only propagation from FairTrackParP → ParP and from FairTrackParH → ParH are available

```
Bool_t Propagate(FairTrackParH* TStart, FairTrackParH* TEnd, Int_t PDG)
Bool_t Propagate(FairTrackParP* TStart, FairTrackParP* TEnd, Int_t PDG)
```

❖ if you are backpropagating

```
Bool_t void setBackProp() {fPropOption="BPE";}
```

I want to propagate to a plane

WHY? *Because this is the most used kind of propagation in pandaroot, in the Kalman fit in particular*

WARNING! When you will use the Kalman fit you will not explicitly see this propagation, since it is hidden in GENFIT call to GEANE, but it is a good example

❖ In your Task, you have to setup the propagator:

```
FairGeanePro *geanePro = new FairGeanePro();
```

❖ Then, you need to set the unit vectors which define your starting and ending planes:

```
// set up the vectors spanning the start plane
TVector3 v1s;
TVector3 v2s;
// set up the origin ...
TVector3 v0e;
// ... and the vectors spanning the end plane
TVector3 v1e;
TVector3 v2e;
```

I want to propagate to a plane

❖ Then you have to communicate to the propagator your intentions

```
geanePro->PropagateFromPlane(v1s, v2s);
geanePro->PropagateToPlane(v0e, v1e, v2e);
```

❖ setup the starting and ending FairTrackParP (required when propagating to plane)

```
FairTrackParP *startPar = new FairTrackParP(opportune ctor);
FairTrackParP *endPar = new FairTrackParP();
```

❖ If you want to go **backward**

```
geanePro->setBackProp();
```

❖ And eventually you can start the actual propagation

```
geanePro->Propagate(startPar, endPar, pdgCode);
```

WARNING! you need to give a mass hypothesis!

I want to propagate to a PCA

***WHY?** Because when I want to propagate to the vertex I need to propagate to PCA, a.k.a POCA, which means **P**oint **Of** **C**losest **A**pproach*

The propagation to the point of closest approach to a space point or to a line is performed through two steps:
- a propagation to a large track length during which the PCA is found
- the actual propagation from the starting point to the PCA is done.

- In your Task, you have to setup the propagator:

AS BEFORE

```
FairGeanePro *geanePro = new FairGeanePro();
```

- Choose whether to propagate to the poca to a point (1) or to a line (2)

```
geanePro->PropagateToPCA(1,dir);
```

```
geanePro->PropagateToPCA(2,dir);
```

- set to which point or line:

```
TVector3 spacePoint;
geanePro->SetPoint(point);
```

```
TVector3 ex1, ex2
geanePro->SetWire(ex1, ex2);
```

I want to propagate to a PCA

AS BEFORE

❖ setup the starting and ending FairTrackParH (required when propagating to pca)

```
FairTrackParH *startHel = new FairTrackParH(opportune ctor);
FairTrackParH *endHel = new FairTrackParH();
```

❖ And eventually you can start the actual propagation

AS BEFORE

```
geanePro->Propagate(startHel, endHel, pdgCode);
```

WARNING! you need to give a mass hypothesis!

GEANE Summary

To use GEANE you need to:

1. In the **macro**:
   - ❖ Create and add to the FairRunAna the GEANE Task:

```
FairGeane *geane = new FairGeane();
fRun->AddTask(geane);
```

2. In the **task** which will use GEANE:
   - ❖ Create the propagator FairGeanePro

```
FairGeanePro *geanePro = new FairGeanePro();
```

   - ❖ Set the propagation type
   - ❖ Write the correct track representation FairTrackParP/H
   - ❖ Actually propagate the particle

YOUR TURN!

EXERCISE 3

try to extrapolate the PR PndTrack to the poca to (0, 0, 0) to get the starting point for the Kalman fit

EXERCISE 4

try to extrapolate the Kalman PndTrack to the poca to (0, 0, 0) to get the initial value of position and momentum

Hints:

INPUT: PndTrack
you have to BACKPROPAGATE!

Summary

- ❖ Introduction
- ❖ Input/Output Data
- ❖ Pattern Recognition
- ❖ Track Fitting
- ❖ Track Extrapolation
- ❖ Some macros…

Efficiency/Resolution Macro

❖ At the end of the tracking procedure we have the PndTrack object.

❖ It contains the track parameters at the first and at the last points (we consider here the first point)

```
for(int itrk = 0; itrk < fTrackArray->GetEntriesFast(); itrk++)

PndTrackID *trkID = (PndTrackID*) fTrackIDArray->At(itrk);
if(trkID->GetCorrTrackID() != 0) continue;

PndTrack *trk = (PndTrack*) fTrackArray->At(itrk);
if(trk->GetFlag() < 0) continue;

FairTrackParP firstpar = trk->GetParamFirst()
TVector3 mom = firstpar.GetMomentum();
```

Where to get some further info

Pattern Recognition

- ❖ MVD Technical Design Report
- ❖ STT Technical Design Report
- ❖ presentations from the Collaboration Meetings/Computing Sessions

GENFIT

http://panda-wiki.gsi.de/cgi-bin/view/Computing/GenFit

GEANE

http://panda-wiki.gsi.de/cgi-bin/view/Comp

… and we are done

Thank you!

Workpackages

pattern recognition

❖ Study the curling tracks

❖ If there you have new ideas, try different solutions for the primary/secondary track finder, particularly for time gaining

GENFIT

❖ Test the new revision in pandaroot, particularly:

- ❖ the deterministic annehaling filter (to eliminate outliers)
- ❖ the RKTrackRep *vs* the GeaneTrackRep

GEANE

❖ study the electron tracking