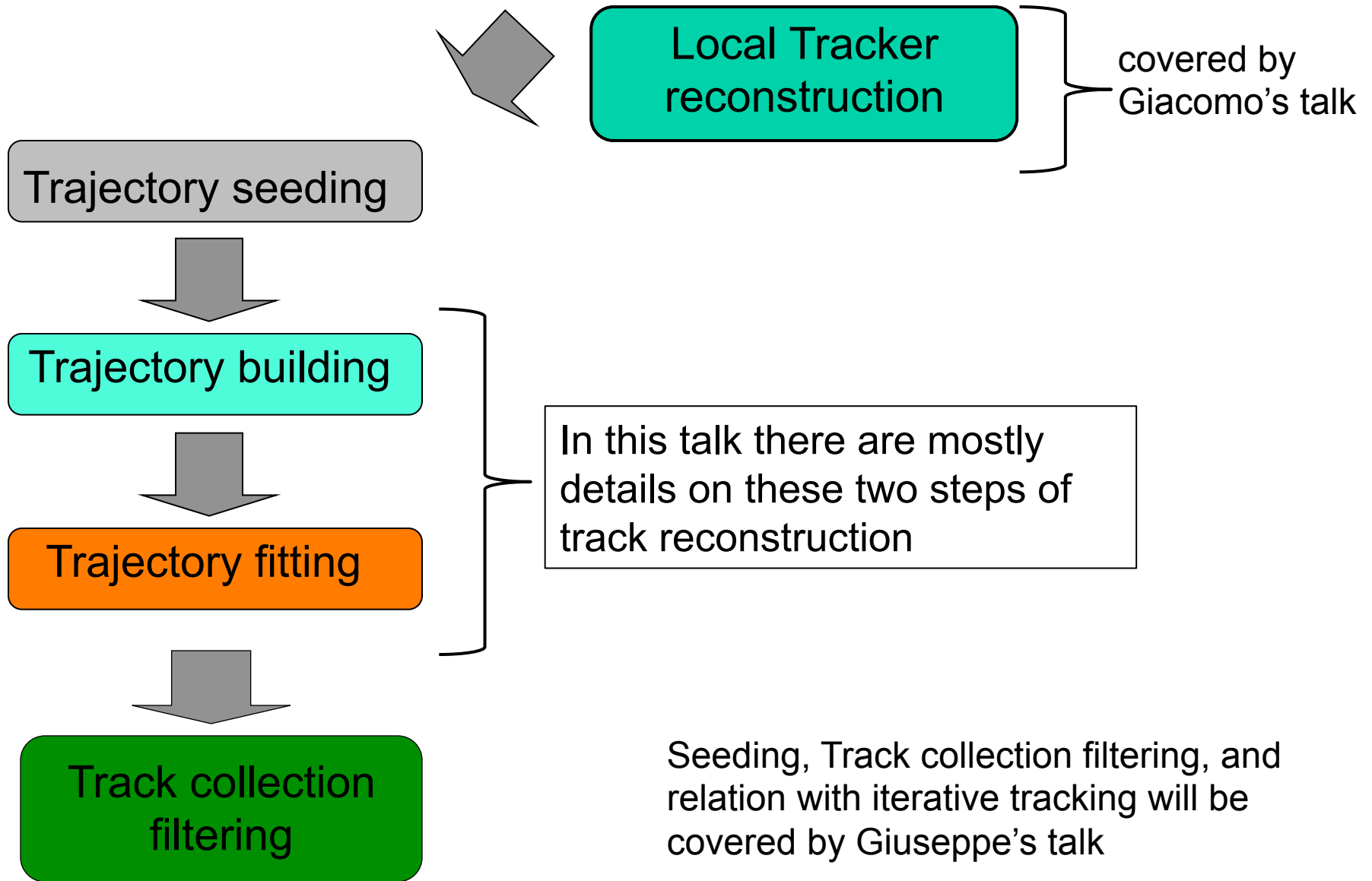


Implementation of Trajectory Building and Track Fitting in CMSSW

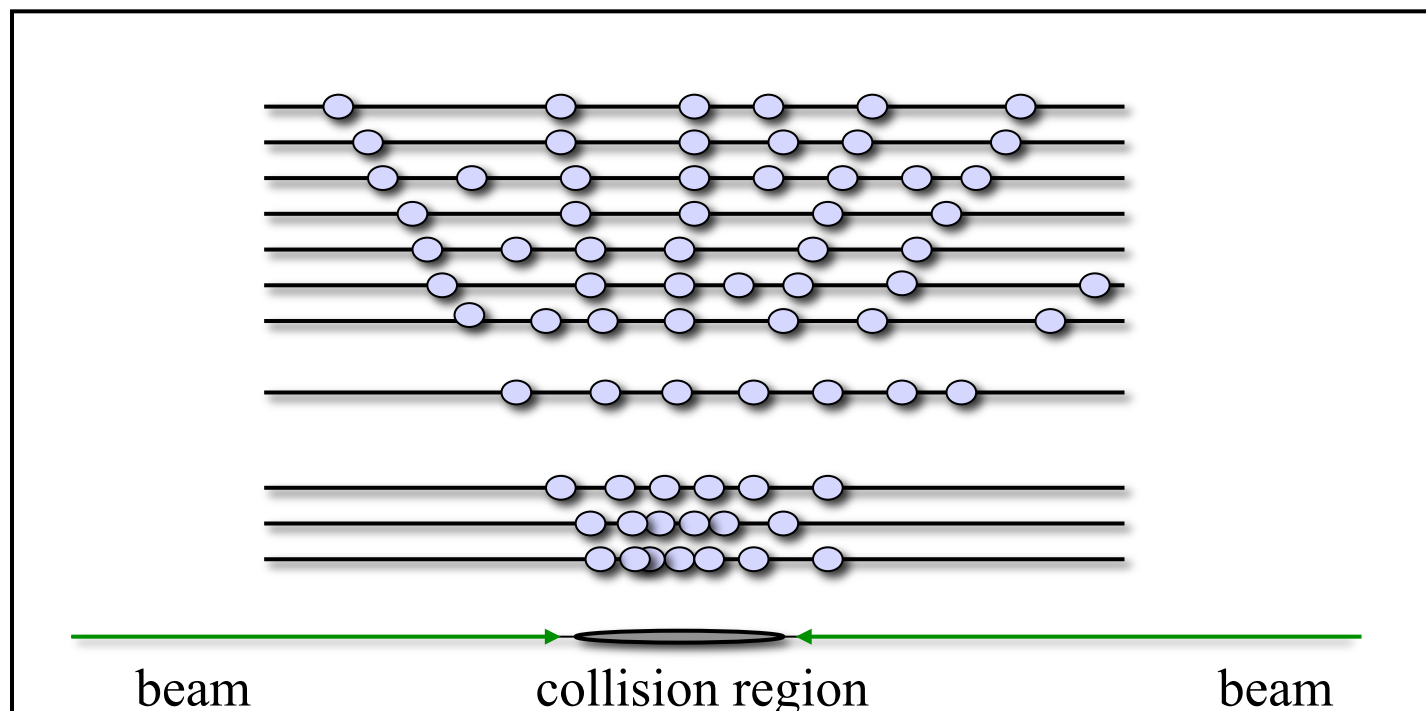
B.Mangano
(ETH Zurich)



Local Tracker reconstruction

Pixel and silicon-strip signals are clustered. Position and corresponding error matrices of each hit are evaluated.

Output is collection of pixel and strip rechits

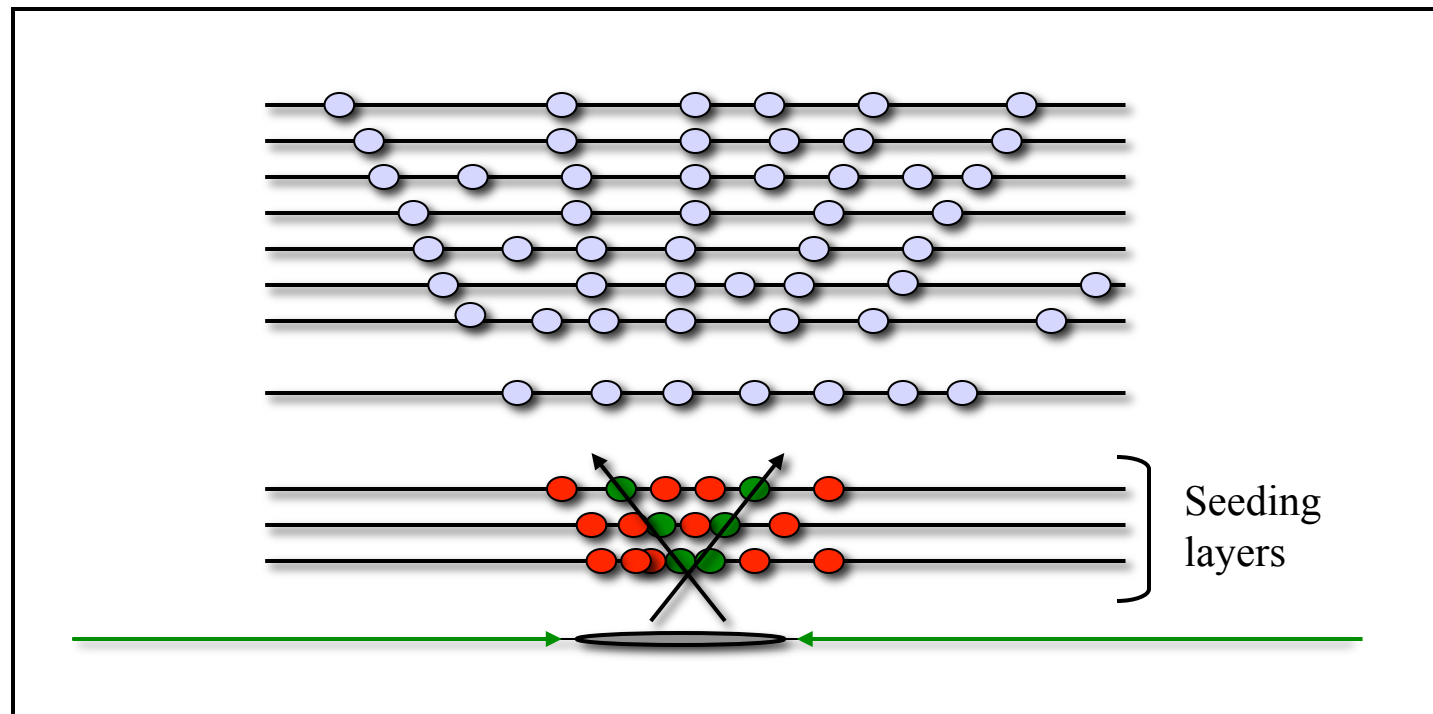


ETH Steps for Track Reconstruction (II)

Trajectory seeding

Initial estimate of trajectory parameters from a small subset of Tracker measurements, i.e. the hits on the *seeding* layers of the detector.

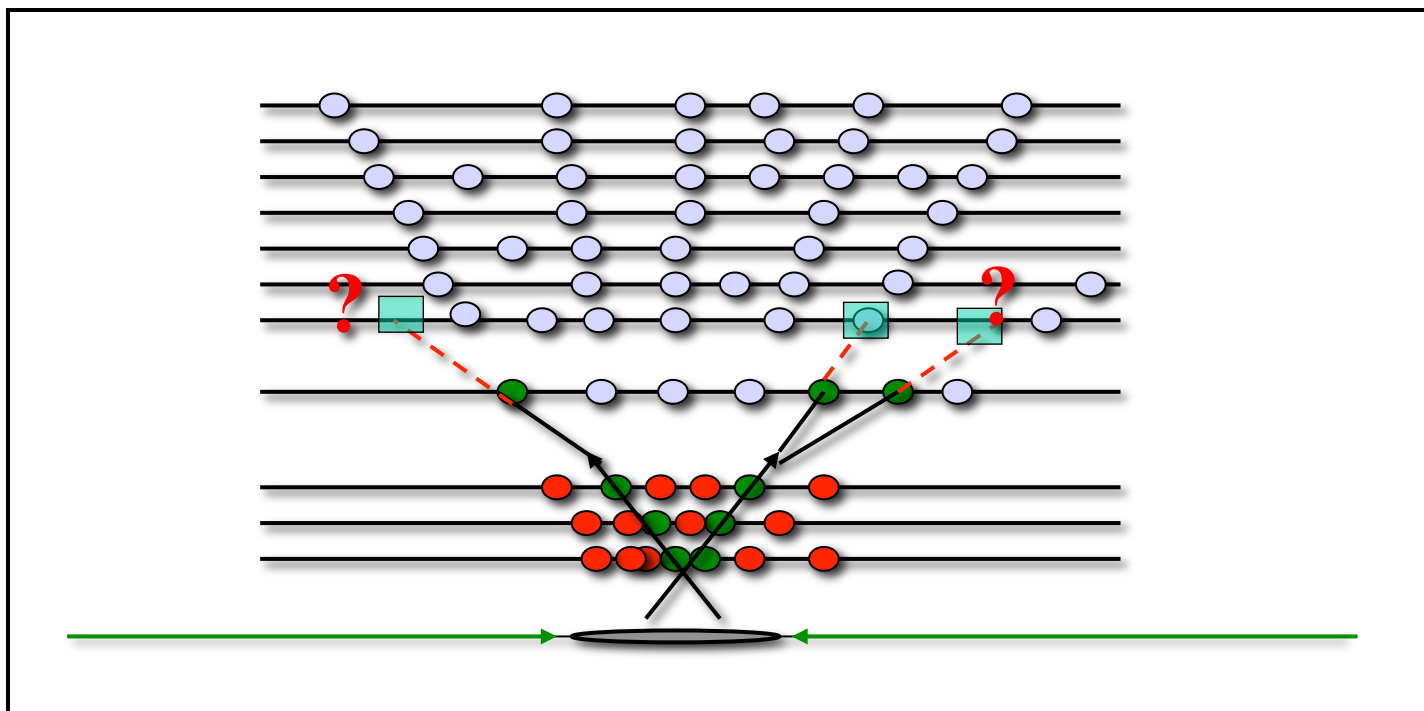
Output is collection of **TrajectorySeed**



Trajectory building

Iterative process which collects all “hits” associated to the same charged particle.

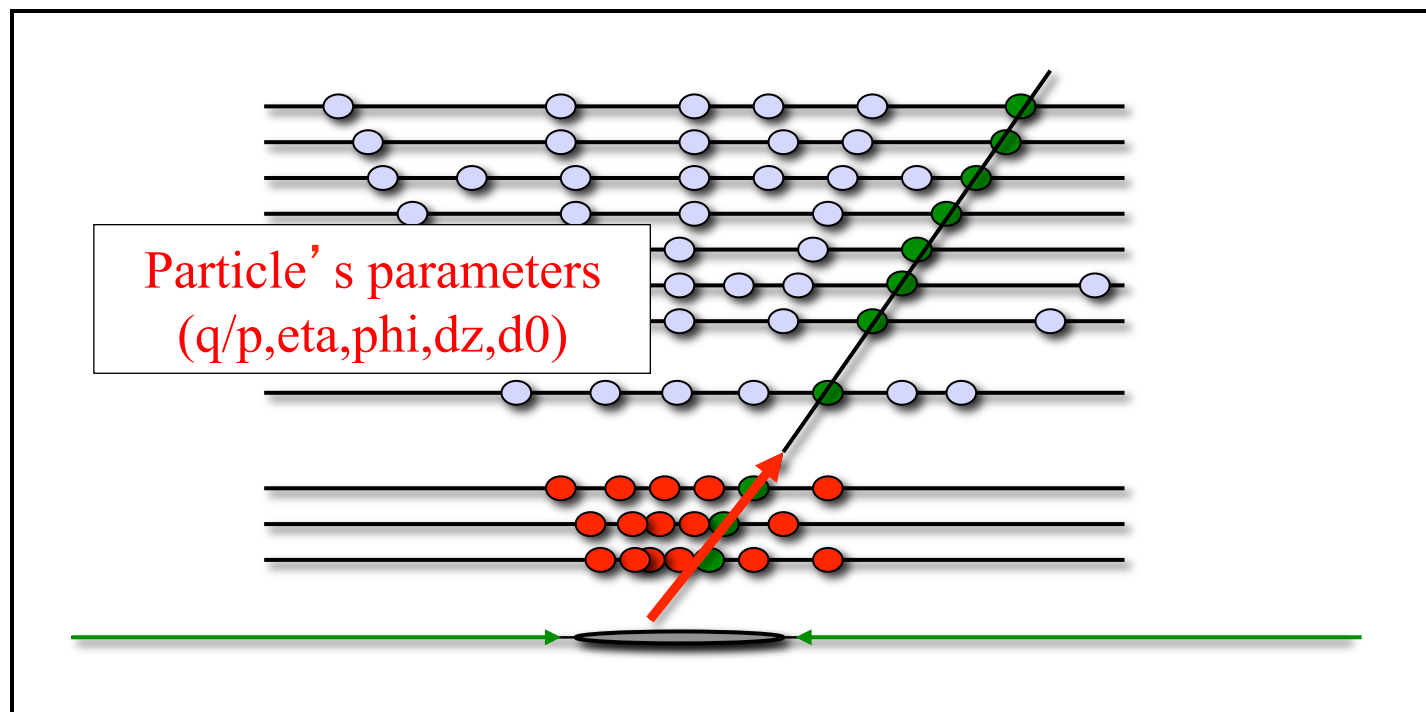
Output is collection of `TrackCandidate`



Trajectory fitting

Estimation of final track parameters from the Kalman Filter+Smoother fit of the full set of measurements associated to the same charged particle.

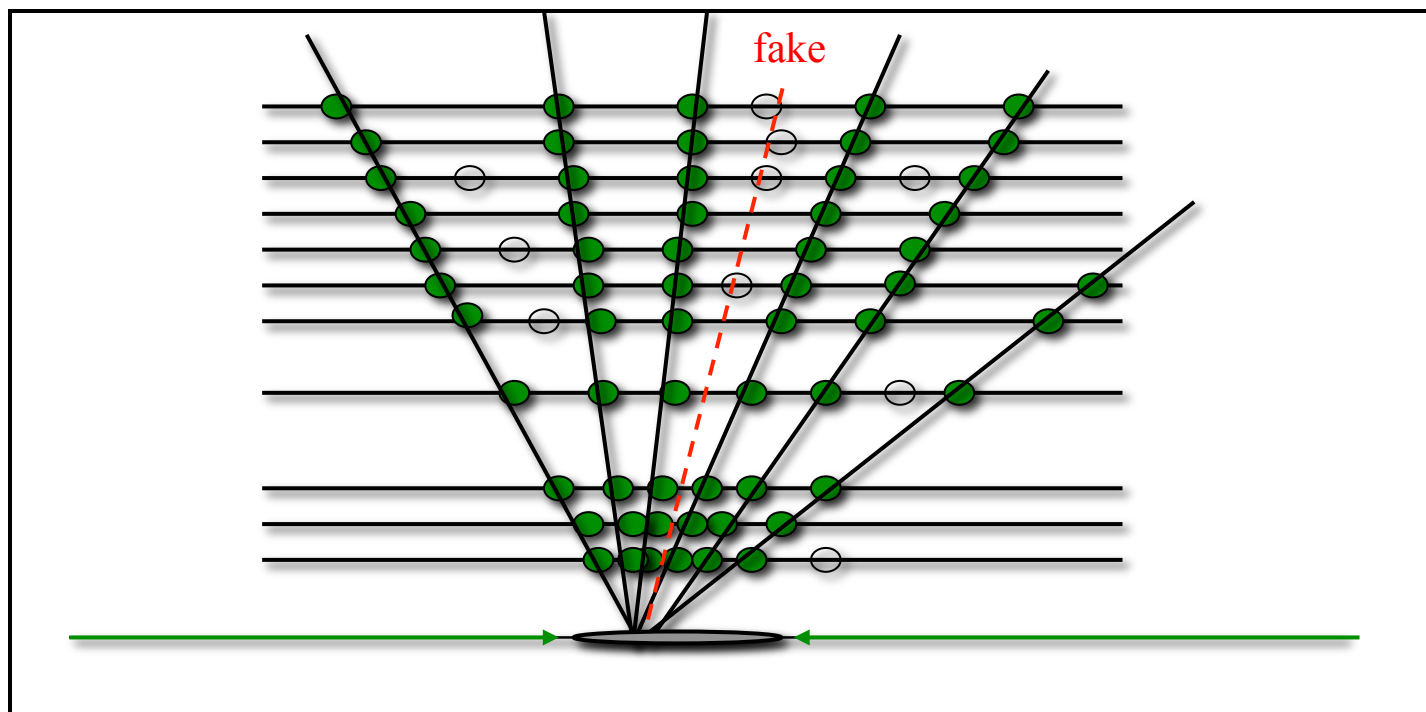
Output is collection of `reco::Track`



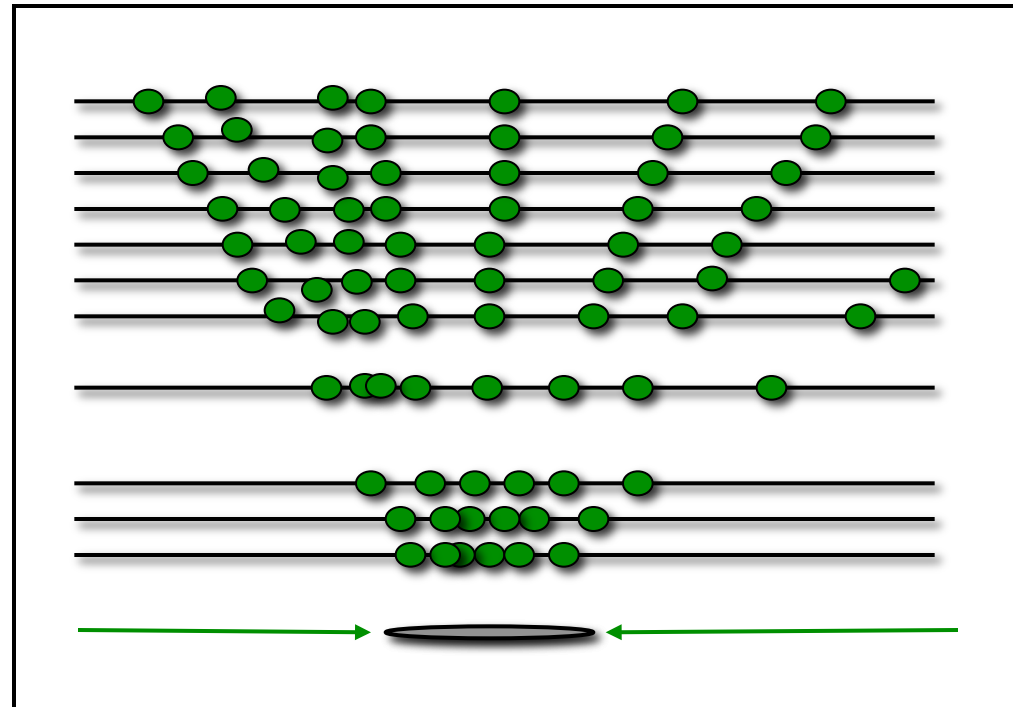
“Cleaning” of the
tracks collection

Removal of fake tracks.

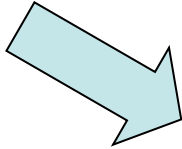
Output is a smaller collection of `reco::Track`



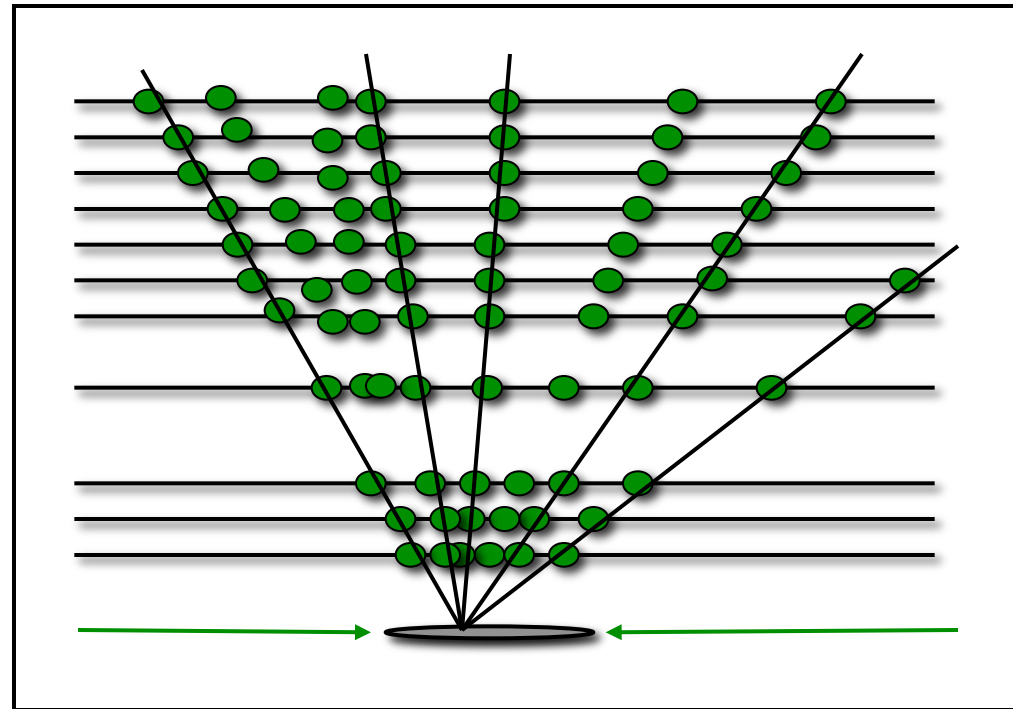
Initial collection of hits

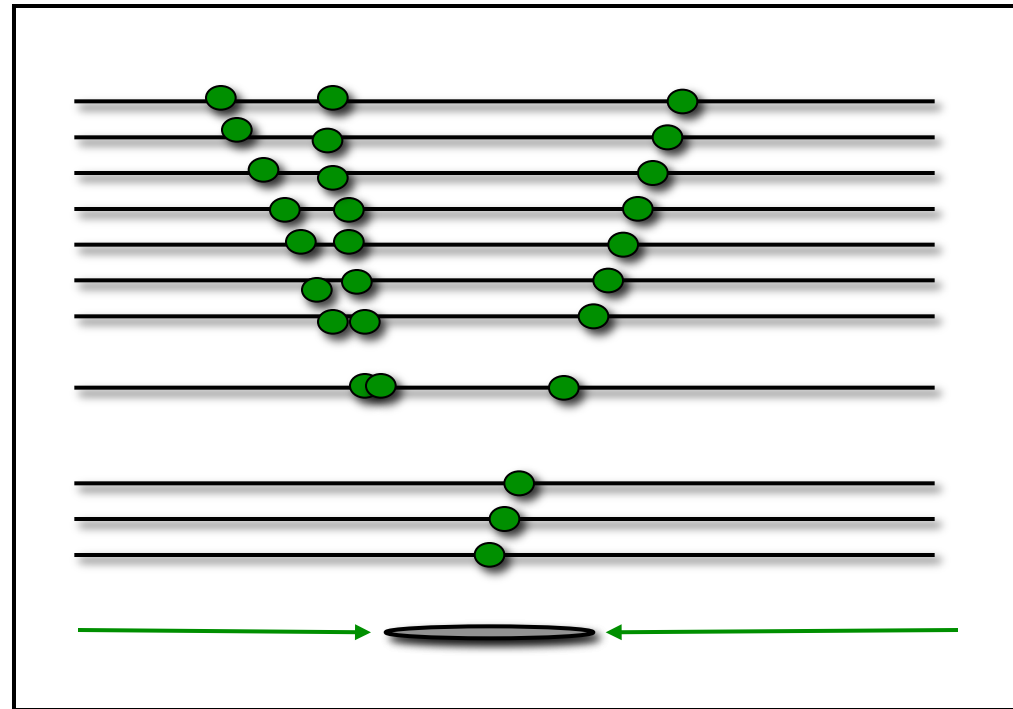
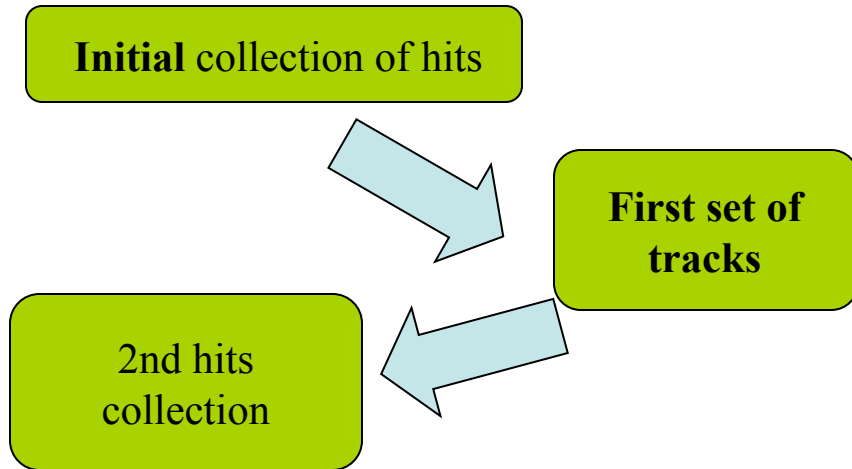


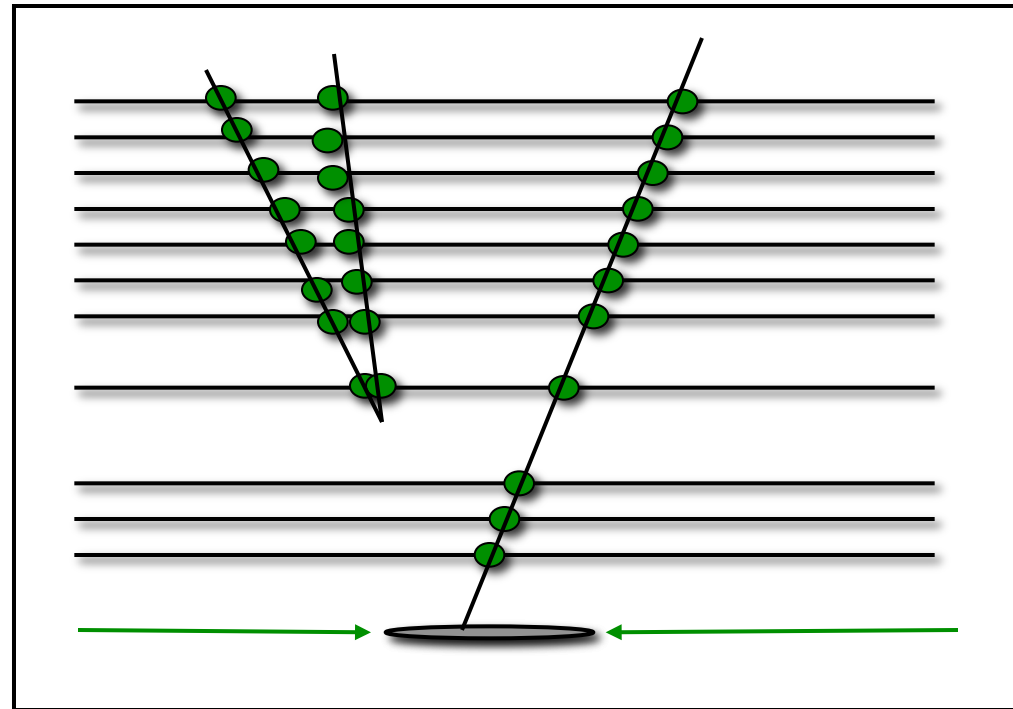
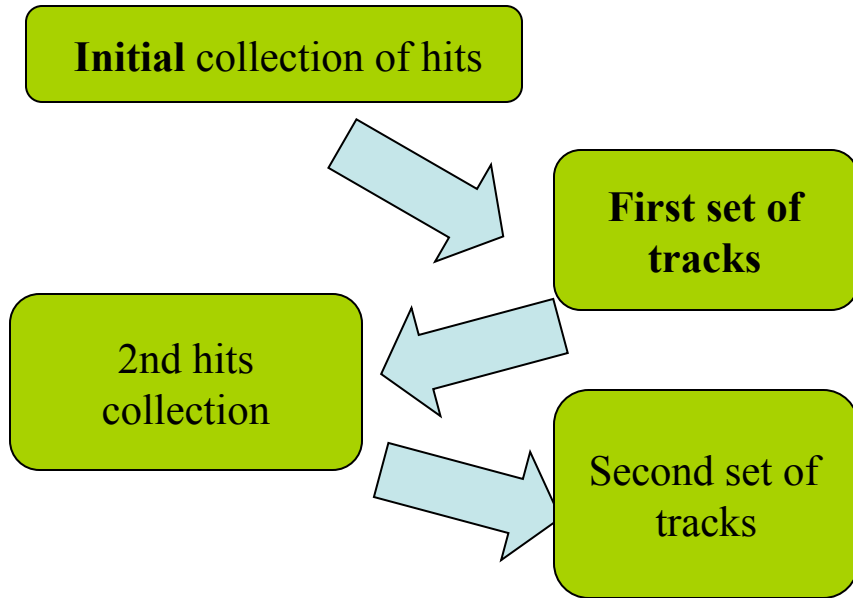
Initial collection of hits

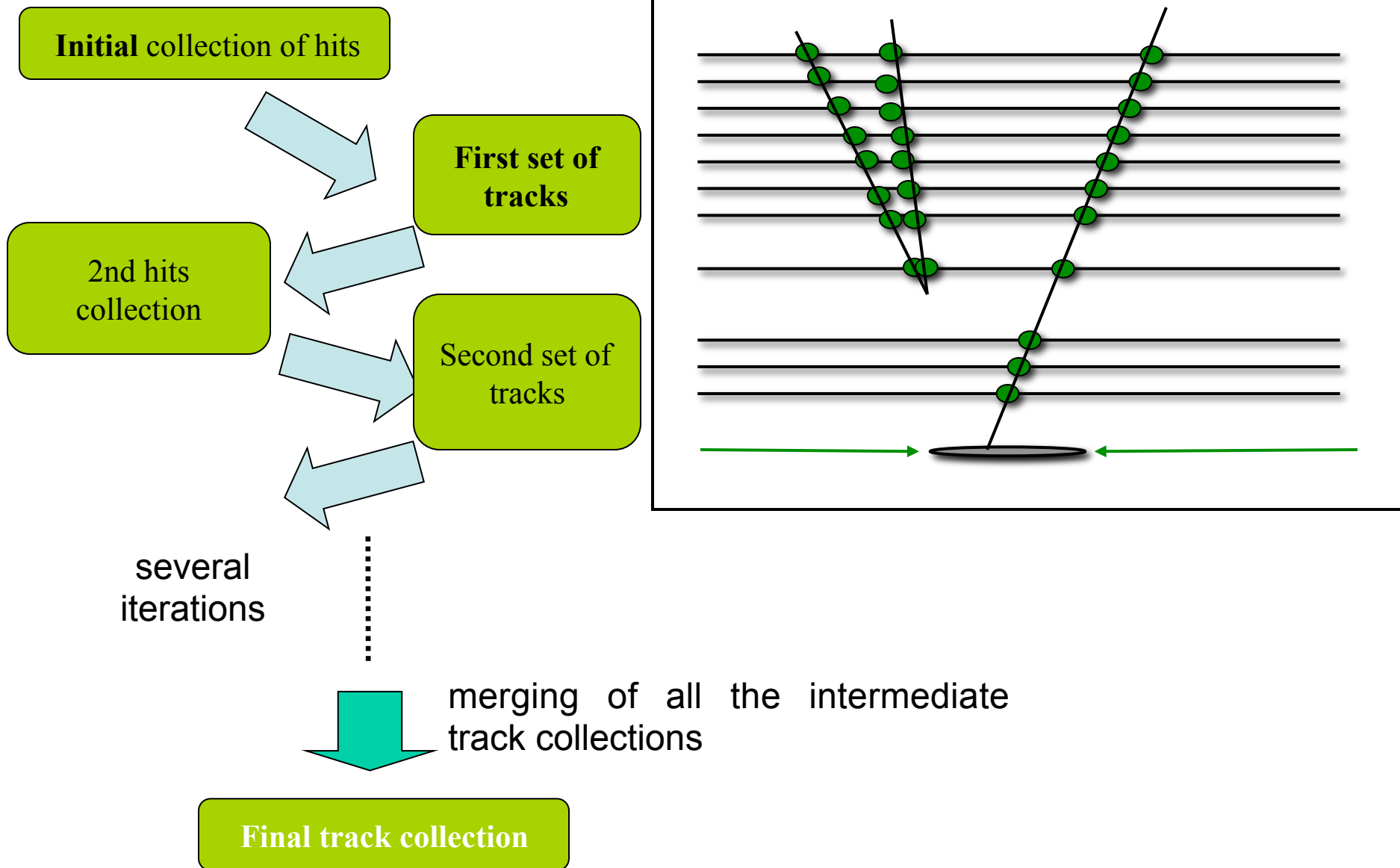


First set of tracks



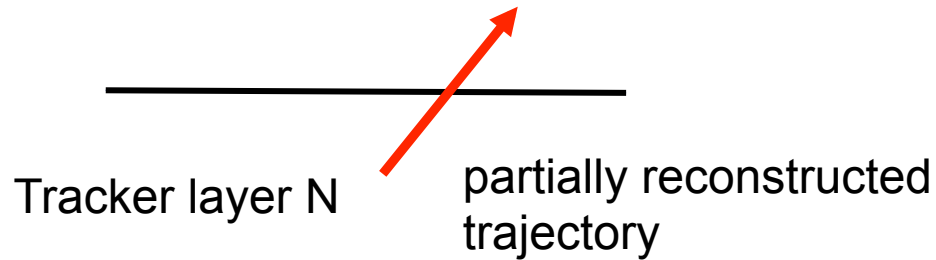






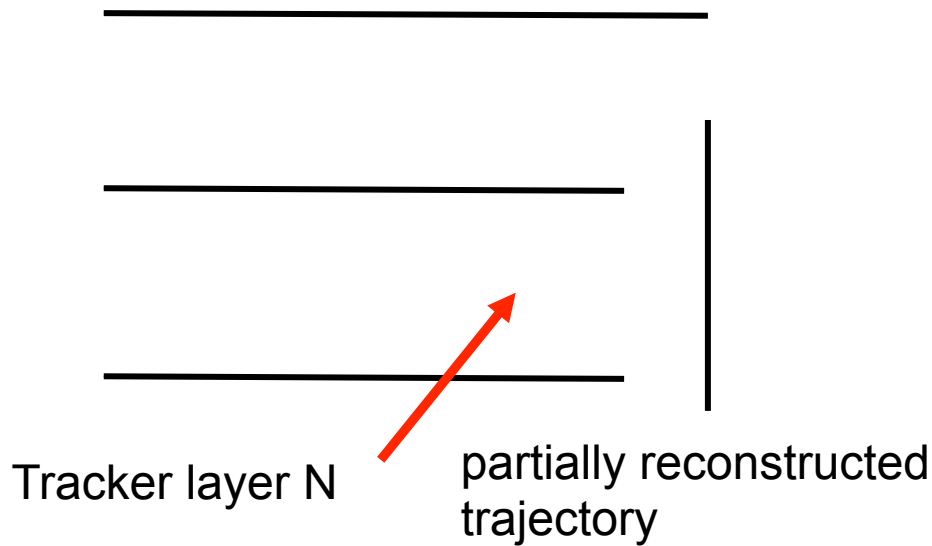
ETH Pattern Reco: smallest logic block

input: trajectory state on Layer N



ETH Pattern Reco: smallest logic block

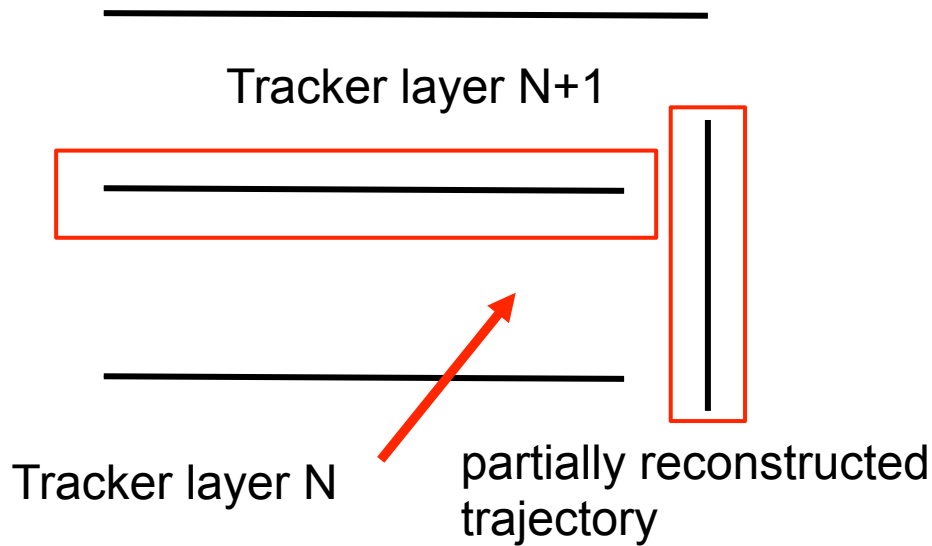
input: trajectory state on Layer N



- 1) find **next** compatible layer (or layers)

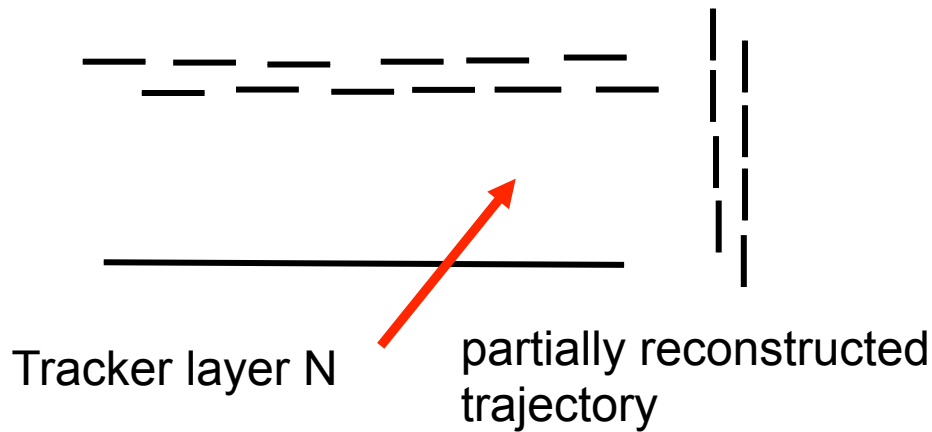
ETH Pattern Reco: smallest logic block

input: trajectory state on Layer N



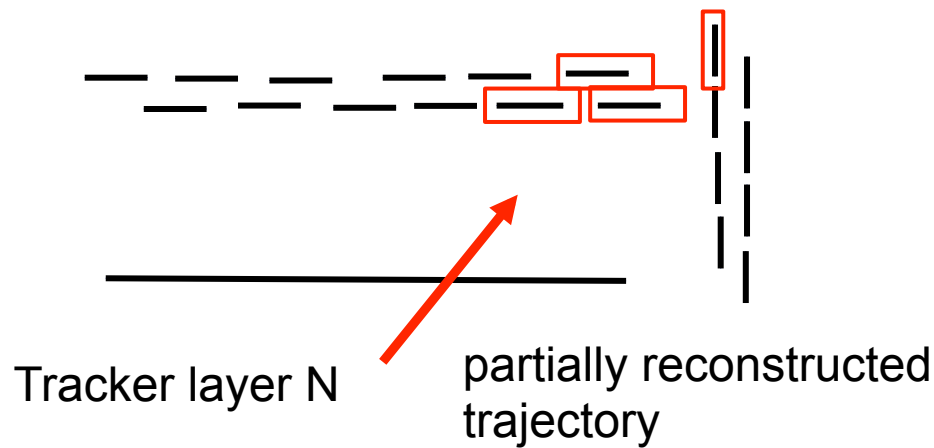
- 1) find **next** compatible layer (or layers)

input: trajectory state on Layer N



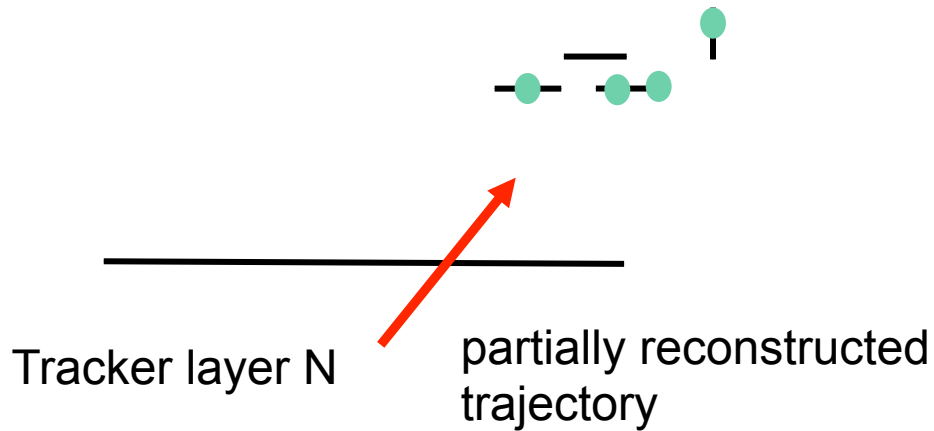
- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1

ETH Pattern Reco: smallest logic block



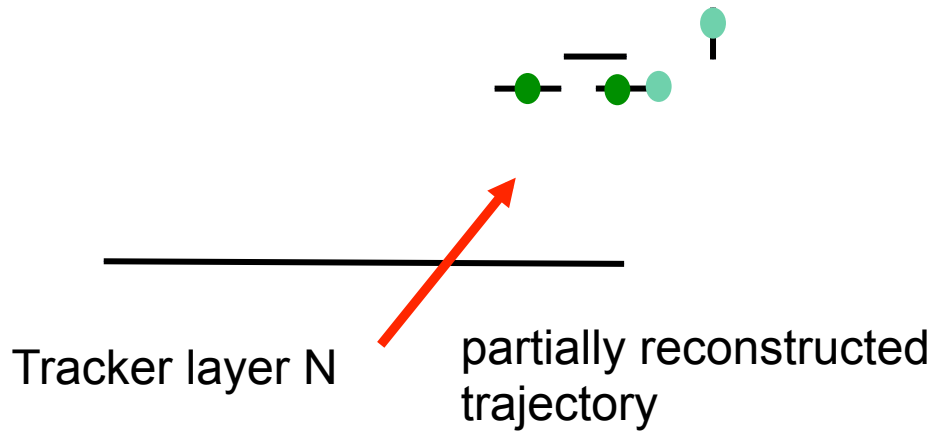
input: trajectory state on Layer N

- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1



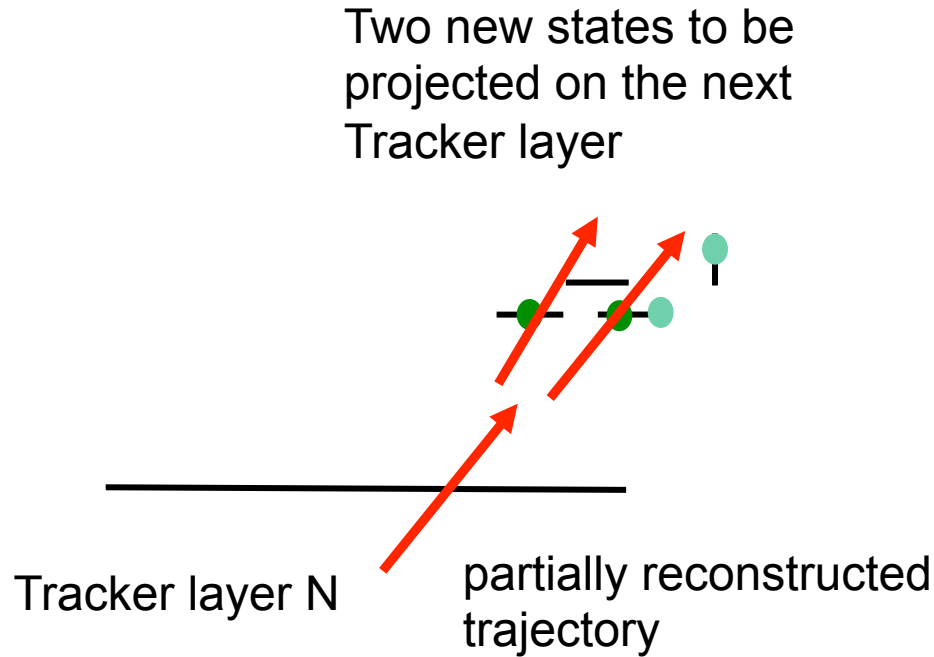
input: trajectory state on Layer N

- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1
- 3) find list of compatible measurements on compatible dets



input: trajectory state on Layer N

- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1
- 3) find list of compatible measurements on compatible dets



input: trajectory state on Layer N

- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1
- 3) find list of compatible hits on compatible dets
- 4) Create new trajectory with update state (pt,eta,phi,..) for each compatible hit

Repeat same 4 steps for each newly created state, until the last (outermost or innermost) layer of the tracker is reached

input: trajectory state on Layer N



- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1
- 3) find list of compatible hits on compatible dets
- 4) Create new trajectory with updated state (pt,eta,phi,..) for each compatible hit

ETH Pattern Reco: smallest logic block

Given a `Trajectory`, it is asked for its last `TrajectoryMeasurement`.

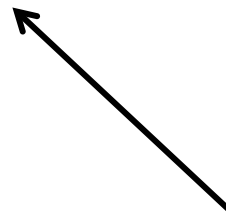


1) find **next** compatible layer (or layers)

The measurement knows on which `DetLayer` it sits.

A `DetLayer` returns the next compatible layer through


```
DetLayer.nextLayers(state,  
                    propagationDirection)
```



This can work if the links among layers have been created: see `NavigationSchool`

For a given `DetLayer`, it is possible to retrieve the list of compatible detectors on the layer using

```
DetLayer.compatibleDets(state,  
    propagator,  
    measurementEstimator)
```

- 1) find **next** compatible layer (or layers)
 - 2) find list of compatible detectors on layers N+1
- 

propagator and measurementEstimator are instances of `Propagator` and `MeasurementEstimator` classes that will be described later.

They are created once and used many times (taken from EventSetup)

ETH Pattern Reco: smallest logic block

`MeasurementTracker` services returns the `MeasurementDet` corresponding to a given `DetId`. The `DetId` is taken from the list of compatible detectors from previous step.

Given a `MeasurementDet`, the list of hits compatible with the trajectory state is obtained with

```
MeasurementDet.fastMeasurements(  
    stateOnDet,  
    propagator,  
    measurementEstimator)
```

The result is a vector of `TrajectoryMeasurement`

The `MeasurementDets` have access to the pixel and strip cluster collections.

- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1
- 3) find list of compatible hits on compatible dets
- 4) Create new trajectory with update state (pt,eta,phi,..) for each compatible hit



Each `TrajectoryMeasurement` contains a hit whose position is estimated using the trajectory state information

Given the `TrajectoryStateOnSurface` that was propagated on the new layer and a compatible `TrajectoryMeasurement`, a new updated state is created by the `KFUpdater`

The KF stands for Kalman Filter.

For GSF tracking, a different updater is used: `GsfMultiStateUpdater`

- 1) find **next** compatible layer (or layers)
- 2) find list of compatible detectors on layers N+1
- 3) find list of compatible hits on compatible dets
- 4) Create new trajectory with update state (pt,eta,phi,..) for each compatible hit



Collections:

- Pixel clusters collection
- Strip clusters collection

Conditions:

- list of bad modules
- list of bad channels
- noisy channels
-

```
graph TD; C[Collections] --> MT[MeasurementTracker]; Co[Conditions] --> MT; MT --> D[Everything cached, at a per-module level, in MeasurementDet instances.];
```

MeasurementTracker

Everything cached, at a per-module level, in **MeasurementDet** instances.

Goal is to provide fast access to per-module information to the rest of the tracking code

The CMSSW EDPproducer making the `TrackCandidates` is the `CkfTrackCandidateMaker`, but the actual pattern recognition task is handled by the `TrajectoryBuilder`.

In first approximation it takes care of calling the 4 instructions of the previous block many times for each trajectory seed that is provided in input.

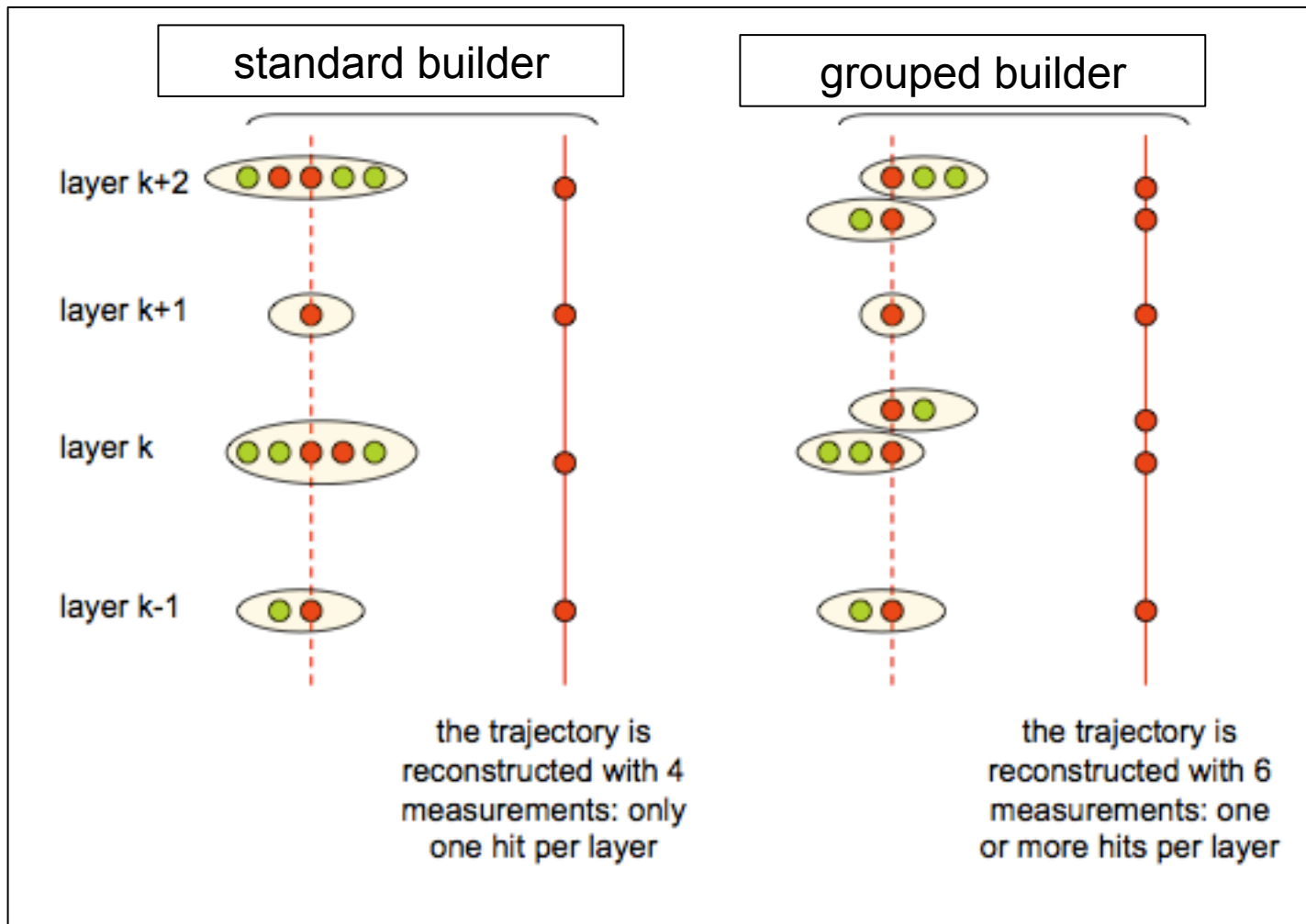
Two flavors of builders are available for standard tracking

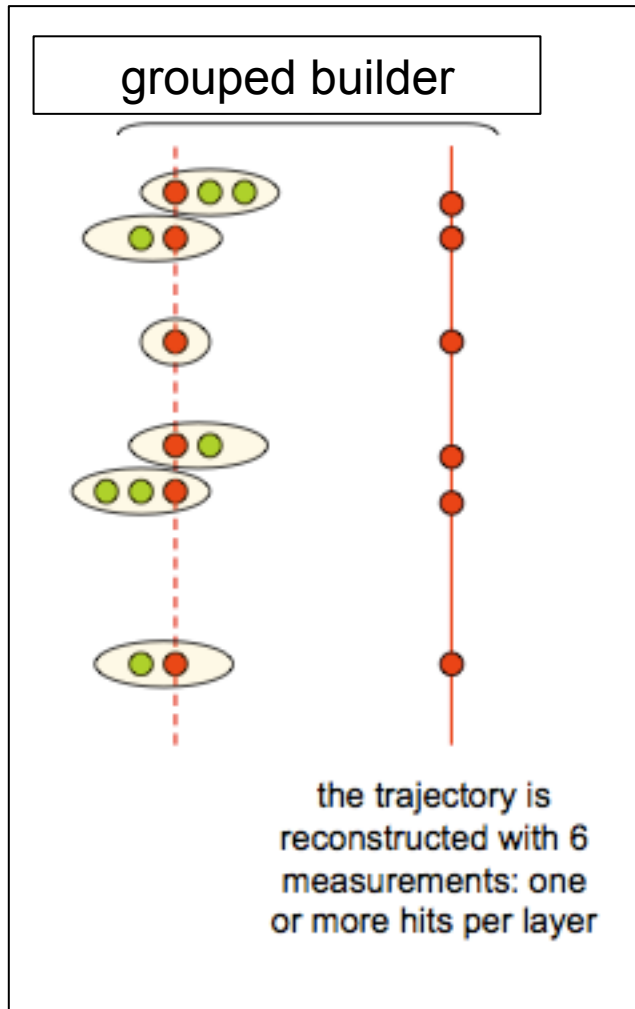
`CkfTrajectoryBuilder`:

simpler, slightly faster, the first that was implemented. It has fewer functionalities.

`GroupCkfTrajectoryBuilder`:

- can build segments from multiple hits within the same tracker layers
- has additional features (like building of cosmic tracks or looping particles) that the simpler builder doesn't have.

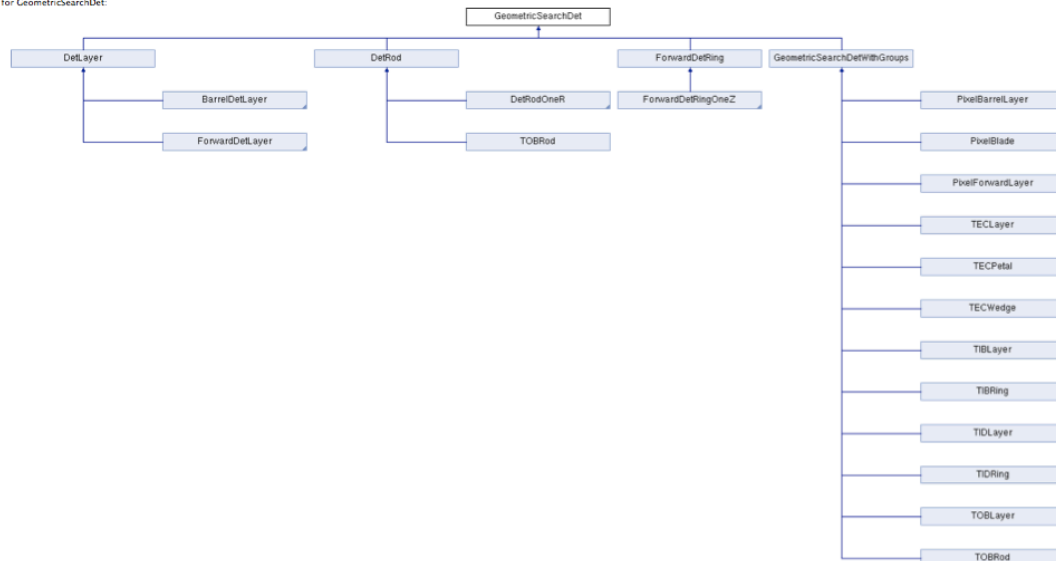




A fast sorting of measurements in groups of mutually exclusive hits is far from trivial.

It is implemented through the hierarchy of classes of the `GeometricSearchDet` tree

```
#include <GeometricSearchDet.h>
Inheritance diagram for GeometricSearchDet:
```



List of all members.

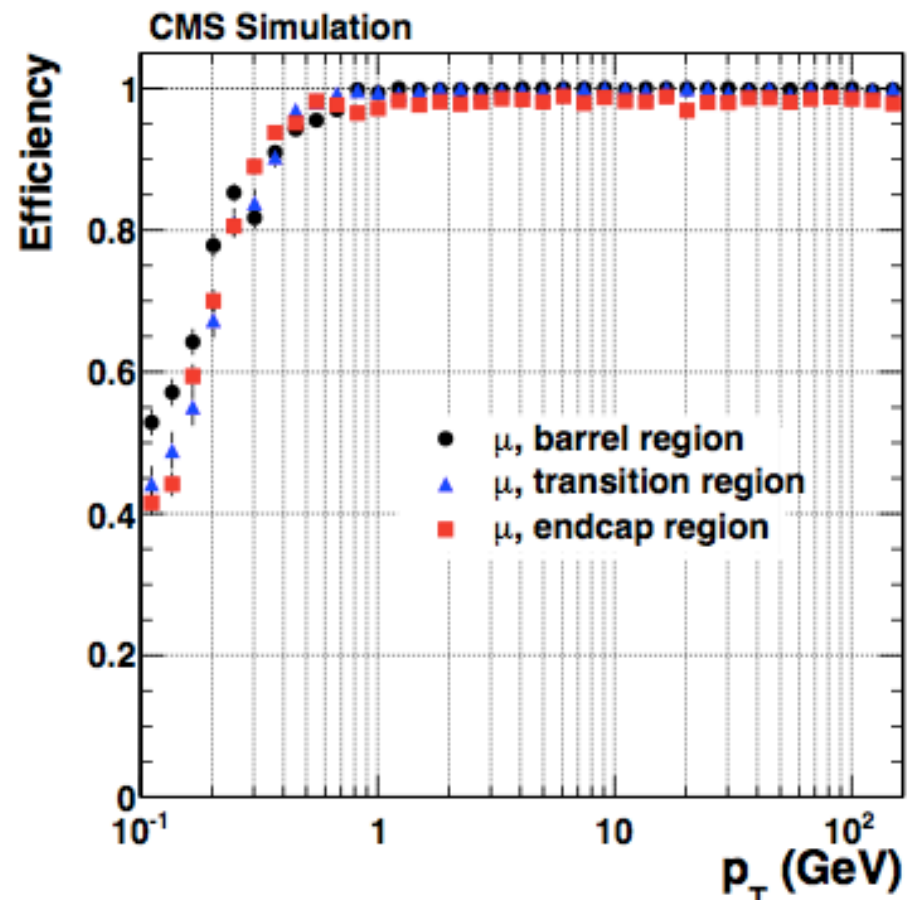
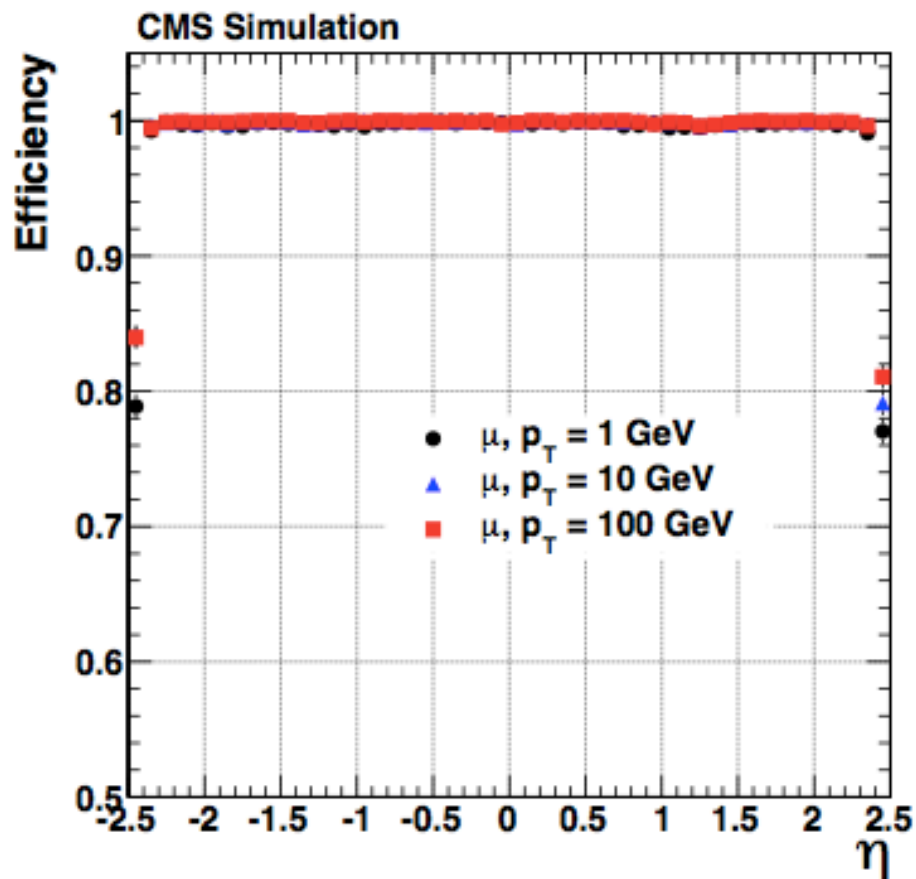
Both the TrackCandidateMaker and the Trajectory builder have a long list of parameters and input “services” that can be configured according to efficiency/speed needs. Among the more important:

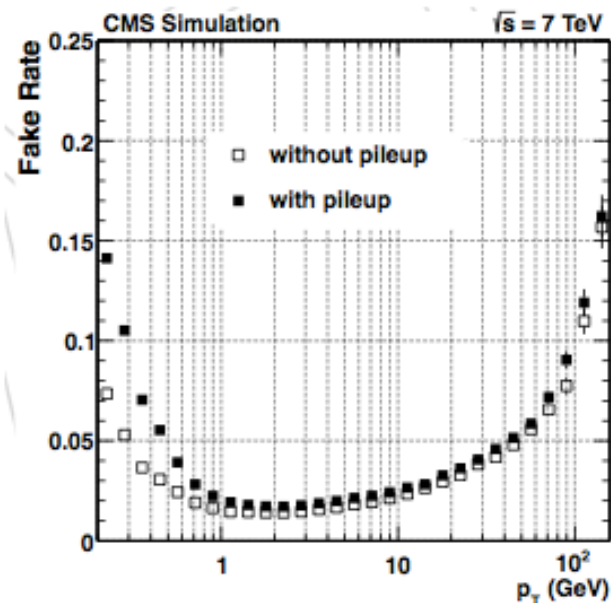
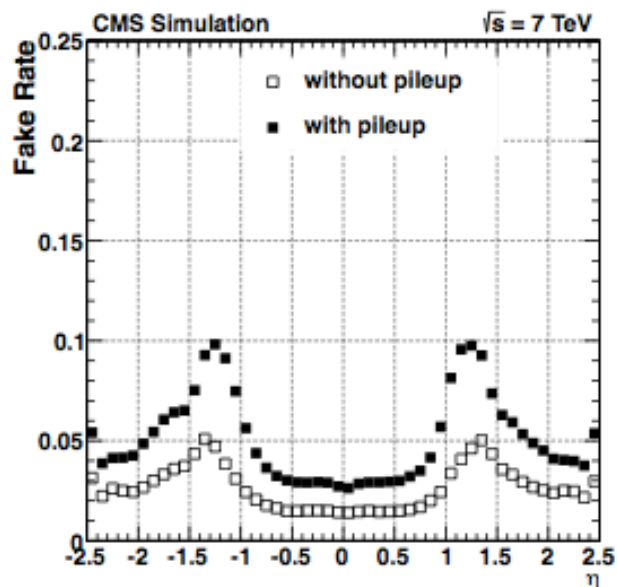
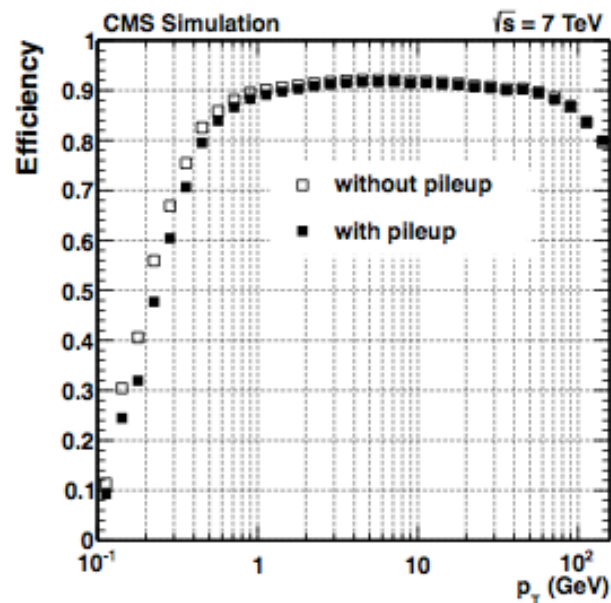
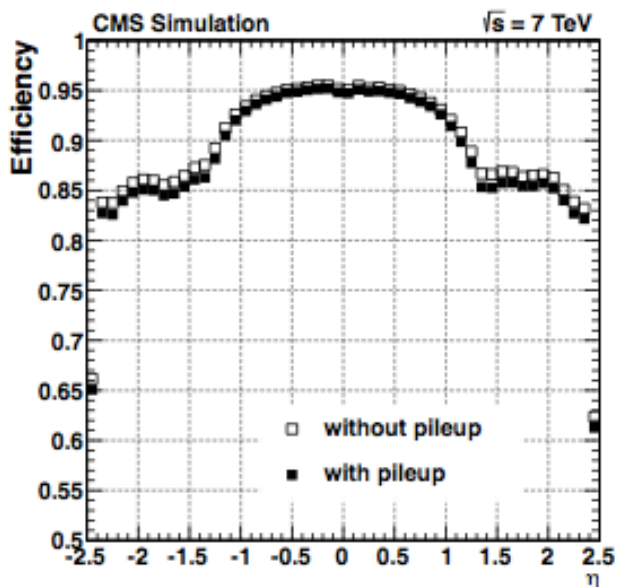
For the CkfTrackCandidateMaker:

- type of builder ([TrajectoryBuilder](#))
- type of trajectory cleaner ([TrajectoryCleaner](#))
- type of seed cleaner ([RedundantSeedCleaner](#))
- [doSeedingRegionRebuilding](#)

For the TrajectoryBuilder:

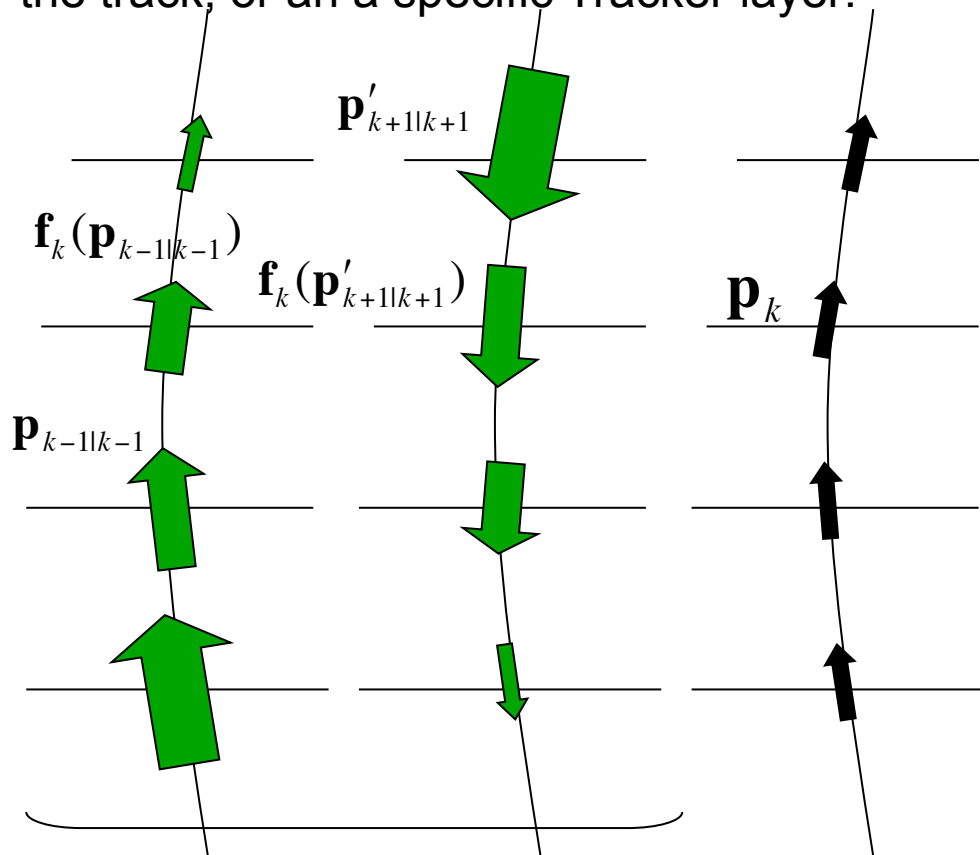
- type of Propagator([propagatorAlong](#) and [propagatorOpposite](#))
- max number of candidates per seed propagated to the next layer ([maxCand](#))
- [lostHitPenalty](#)
- estimator for hit-trajectory compatibility ([Chi2](#))
- trajectory state updator ([updator](#))





Once the set of hits that defines a trajectory is identified, the next step is to fit the positions of the hits to extract the particle parameters: at vertex, at the end of the track, or at a specific Tracker layer:

Once the set of hits that defines a trajectory are identified, the next step is to fit the positions of the hits to extract the particle parameters: at vertex, at the end of the track, or at a specific Tracker layer:



Both in-out filter and out-in one are run

$$\mathbf{p}_k = \mathbf{p}_{k|k} \oplus \mathbf{f}_k(\mathbf{p}'_{k+1|k+1})$$

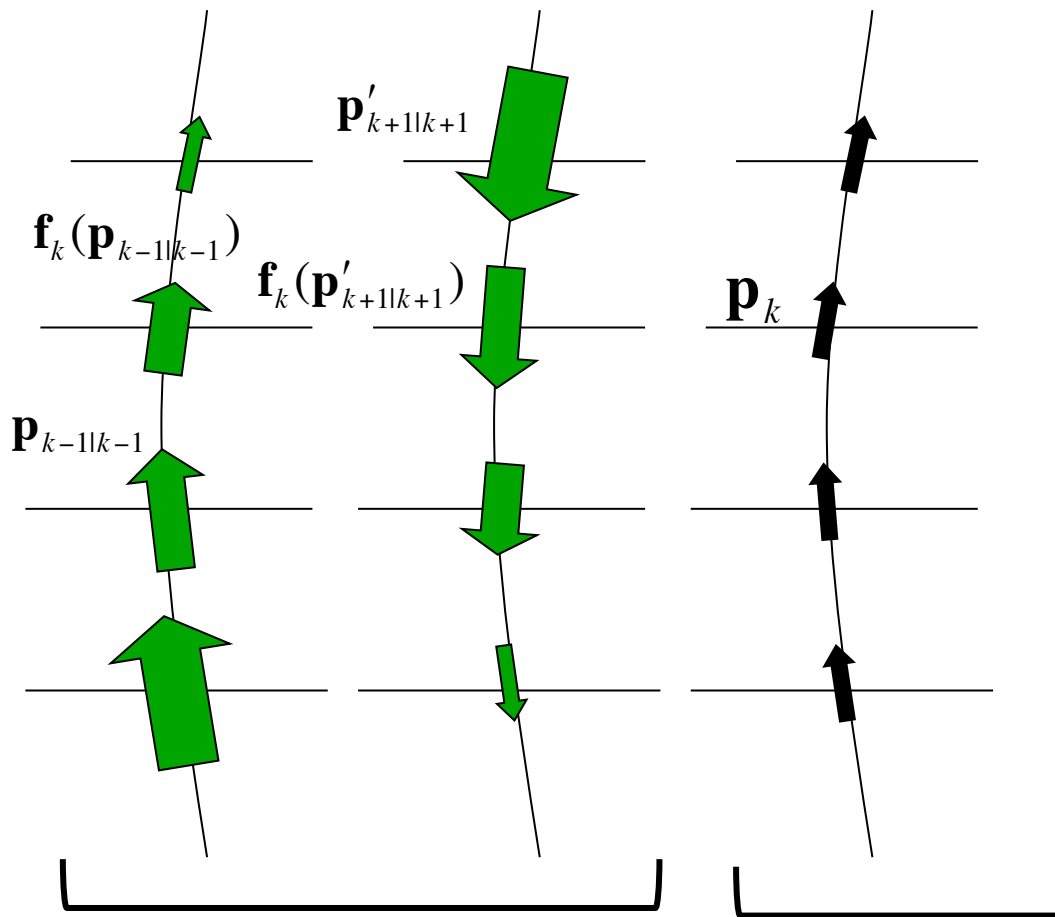
A statistically correct *weighted mean*: **Kalman smoother**

$\mathbf{p}_{k|k}$
Contains information from measurements: 1,2,...,k

$\mathbf{p}'_{k+1|k+1}$
Contains information from measurements: n,n-1,...,k+1

\mathbf{p}_k
Contain the full information. All measurement from 1 to n are used.

All math details in book from R.Fruhwith: "Data Analysis Techniques for High-Energy Physics"



Implemented in
[KFTrajectoryFitter](#) class

Implemented in [KFTrajectorySmoother](#)
 and [KFFittingSmoother](#) classes

This TrajectorySmoother
 also implements **the**
rejection of “outlier” hits.

Each time an outlier is
 identified, it is removed
 from the list of hits and the
 candidate is re-fit.

Process repeated iteratively
 until all hits have a good
 compatibility χ^2 .

If trajectory is too short or
 broken, the candidate is
 rejected

Both during Trajectory Building and Track fitting the trajectory parameters (and the corresponding errors) are propagated between different planes of the Tracker.

This task is performed by services derived from the [Propagator](#) class.

For Tracker tracking, two flavors are used:

1) [AnalyticalPropagator](#):

- use analytical formula
- assume uniform magnetic field between origin and destination surfaces

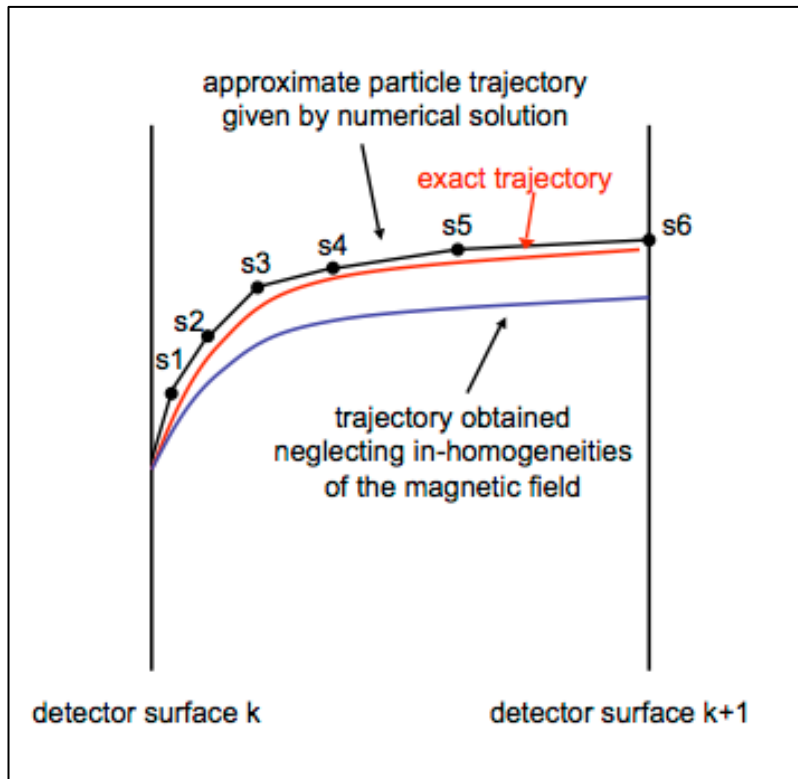
2) [RungeKuttaPropagator](#):

- uses numerical integration of the equations of motion
- handles not uniform magnetic field

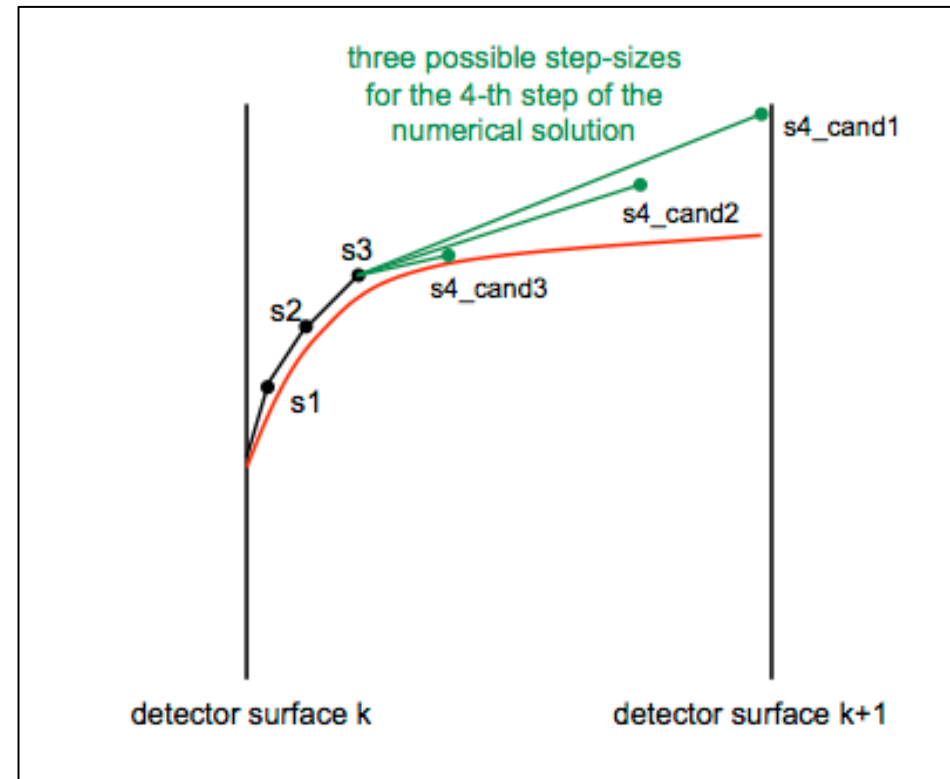
3) [PropagatorWithMaterial](#)

- uses 1) or 2) for propagation from origin to destination surface
- applies expected material effects in a single shot at destination surface using the [MaterialEffectsUpdater](#) class

RungeKutta propagator
performs a **numerical**
integration

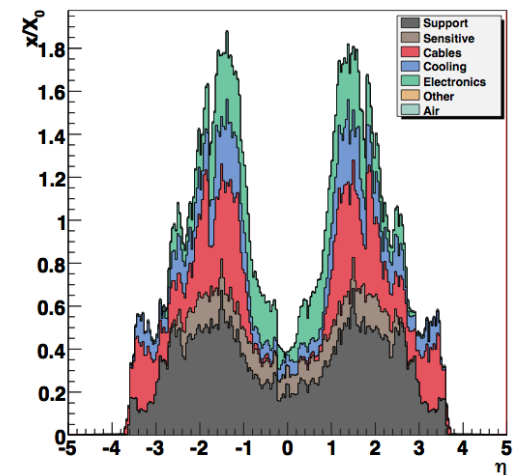
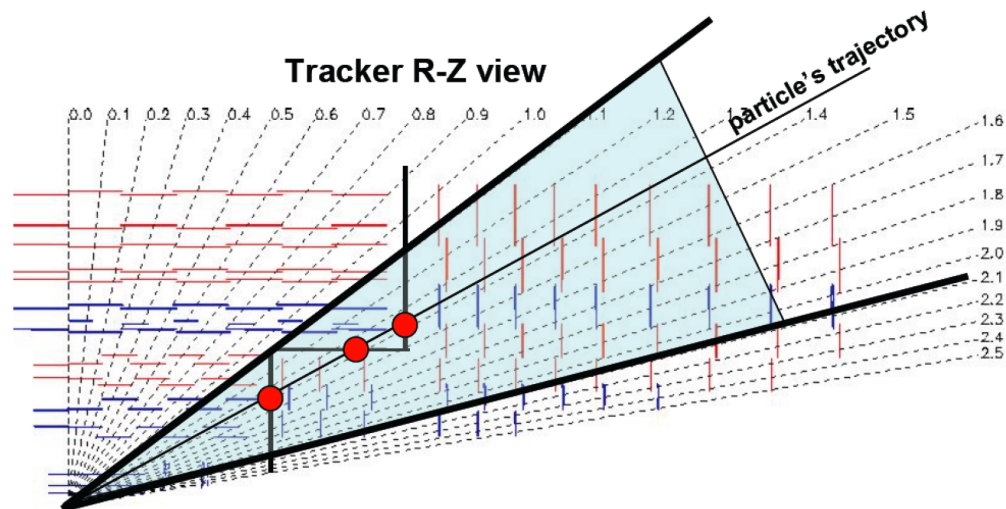


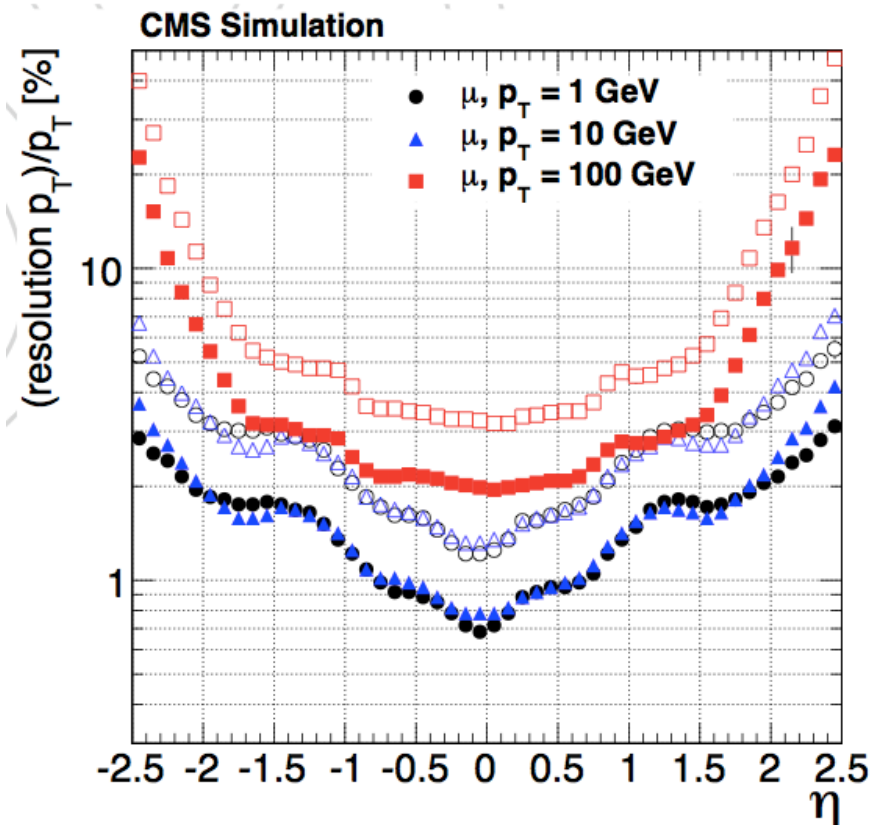
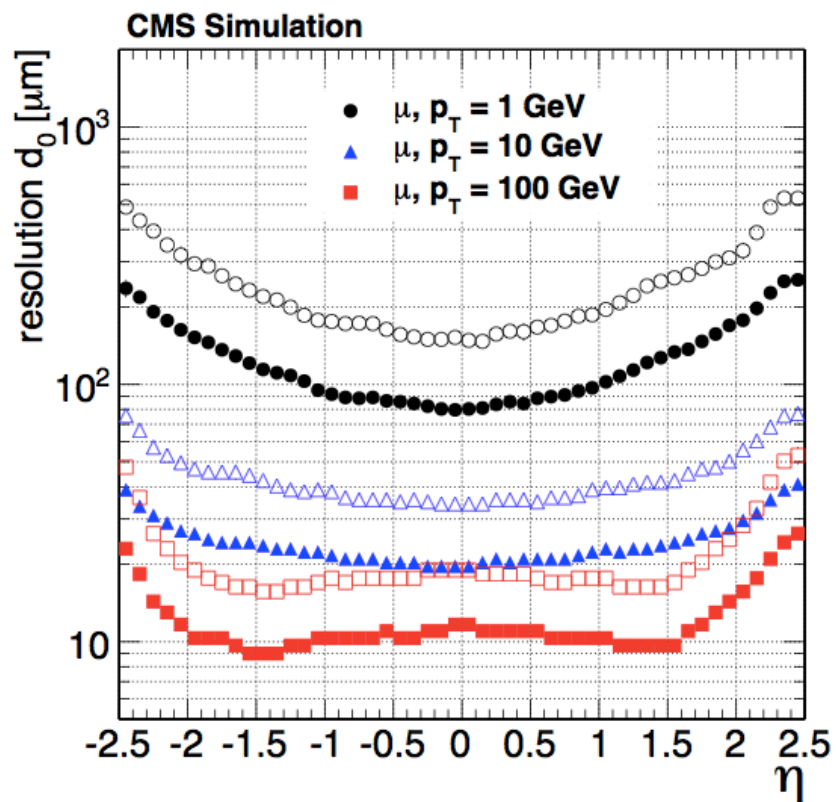
RungeKutta propagator is **adaptive**:
the length of integration “step” is
calculated on the fly depending on an
estimate of the integration precision

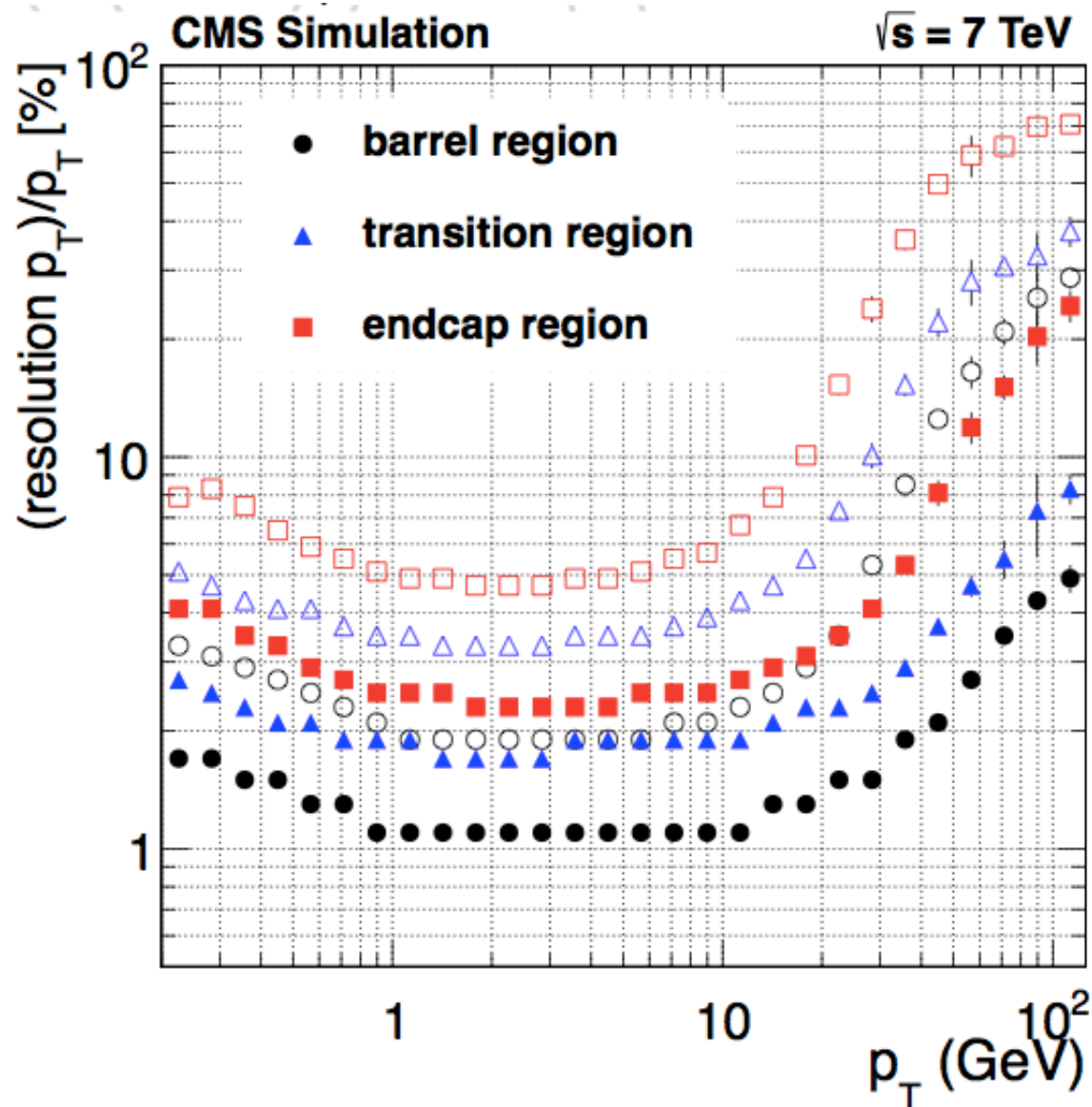


PropagatorWithMaterial logic assumes all the material between origin and destination surfaces is concentrated in the destination surface. It is a reasonable approximation for propagation from a barrel layer to the next barrel layer, or between two endcap disks. However it has some limits for:

- propagating between two modules within the same tracker layer
- propagating in regions with not uniform material distribution like the barrel-endcap area







Only the tip of the tracking code iceberg was shown.

However, it should be enough to orientate a person that wants to start working on tracking.

Many extra details (and performance plots) are available in a paper which is entering now CWR:

<http://cms.cern.ch/iCMS/analysisadmin/cadi?ancode=TRK-11-001>