# Trajectory

## Extending the simple Helix parameterization of Tracks

Frank Gaede
DESY

ILCSoft Meeting
August 10, 2006

# Outline

- Introduction
- Trajectory class proposal
- Class design
- Discussion
  - B-field
  - geometry
  - persistency
- Conclusion

# Introduction

- traditional helix approximation for charged particle tracks is to simplistic and just wrong (sometimes)
  - energy loss of particle decreases r
  - multiple scattering changes path and direction
  - kinks not described at all
  - B field not homegenous

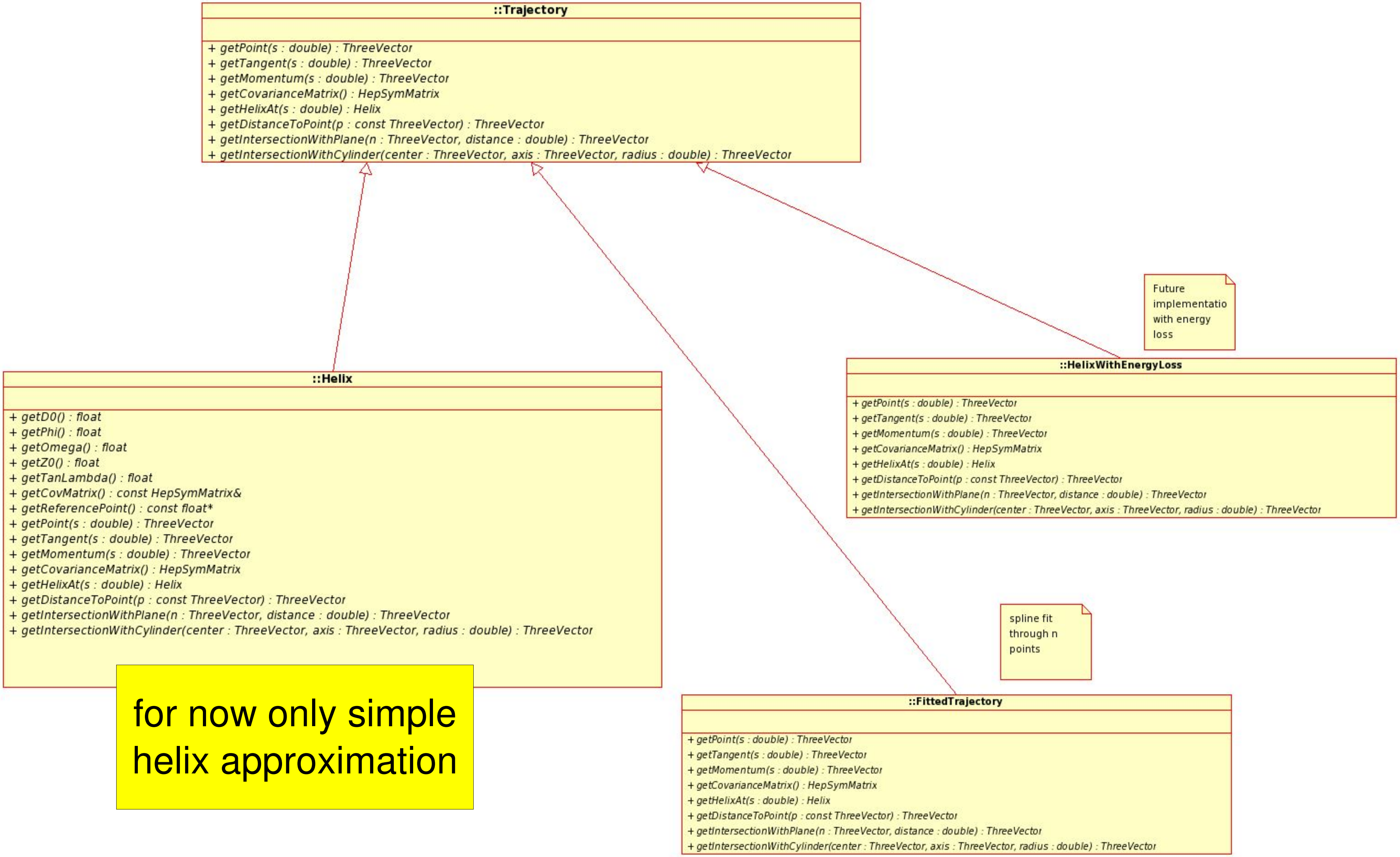
see talk by Benno List

- helix fits are valid only locally – need more than one, e.g. at innermost hit for vertexing and at outermost hit for PFA
- idea: introduce abstract *Trajectory* interface to make it easier in the future to improve the code

# Proposal for Trajectory interface

```
#include <CLHEP/Vector/ThreeVector.h>    // or equivalent from MathMore library
#include <CLHEP/Matrix/SymMatrix.h>      // or equivalent from MathMore library


/** Proposal for a trajectory interface describing a charged particle path in a B field */

class Trajectory {

public:

    /** Point at path length s */
    virtual ThreeVector getPoint(double s) const = 0;

    /** Dirtection at path length s (dx/ds,dy/ds,dz/ds) */
    virtual ThreeVector getTangent(double s) const = 0;

    /** Momentum at path length s  - REQUIRES B(x,y,z) */
    virtual ThreeVector getMomentum(double s) const = 0;

    /** Covariance Matrix of x,y,z,px,py,pz   */
    virtual HepSymMatrix getCovarianceMatrix() const = 0;

    /** Local Helix approximation at s */
    virtual Helix getHelixAt(double s) const = 0;

    /** Distance vector to point */
    virtual ThreeVector getDistanceToPoint(const ThreeVector p)  const = 0;

    /** Closest intersection point with plane - (nan,nan,nan) if none */
    virtual ThreeVector getIntersectionWithPlane(ThreeVector n, double distance) const = 0 ;


    /** Closest intersection point with cylinder - (nan,nan,nan) if none */
    virtual  ThreeVector getIntersectionWithCylinder(ThreeVector center,
                                                     ThreeVector axis,
                                                     double radius) const = 0;

}; // class
```

# Trajectory implementations




for now only simple helix approximation

# B-field

- computing the momentum requires the exact B-field at a given point, e.g.:
  - *ThreeVector b = Bfield.at(  traj.getPosition( s ) )  ;*
  - *ThreeVector p = traj.getMomentum( s , b ) ;*
- or
  - *Trajectory traj(  Bfield ) ;*
  - *ThreeVector p = traj.getMomentum( s ) ;*

need base class for fields, eg:

```
struct ThreeVectorField{
  virtual 3Vec at( 3Vec p) =0 ;
}
```

- issues:
  - computing momentum in arbitrary field not trivial
    - probably OK in almost homogenous field
  - makes code a bit more complicated
    - use z component at local point p :  *double b = Bfield.at( p ).z() ;*
- advantage:
  - code already prepared for introduction of complete field map

# geometry

- current geometry functions assume simplest shapes/surfaces like planes and cylinders:
  - *ThreeVector getIntersectionWithPlane(ThreeVector n, double distance) ;*
  - *ThreeVector getIntersectionWithCylinder(ThreeVector center, ThreeVector axis, double radius) ;*
- this does not describe real detectors with 'arbitrary surfaces' (sagging, missalignement,...)
- however even though it is in principle straight forward to declare a more abstract interface the implementation (computation) is not
  - -> stay with simple shapes for the time being !?

# energy loss - material

- more elaborate implementations of the Trajectory need to take energy loss into account
  - computation possible, provided material at every point is available, e.g GEAR.pointProperties
    - could use CGAGear, i.e. geant4 -> slow, complicated....
  - depends on particle mass (PID hypotheses)
    - *Trajectory traj( Bfield , MUON ) ;*
  - need more elaborate geometry system
    - possibly later this year with SLAC group
  - for now: use simple helix !?

# persistency I

- the current Track class in LCIO uses a Helix parameterization
  - helix fits are valid only locally – need  more than one, e.g. at innermost hit for vertexing and at outermost hit for PFA
- could create a Trajectory off one Helix fit:
    - *Track* trk = dynamic_cast<Track*> ( col->getElementAt( i ) ) ;*
    - *Trajectory traj(  Bfield , trk ) ;*
  - however extrapolation along s only valid in vicinity of reference point
    - swimming only reasonable towards region where not meassured, e.g.
      *ThreeVector p = traj( - |s| )  ;  // for innermost hit (vertexing)*
- -> Trajectories will not be made persistent in current LCIO/Marlin framework

# persistency II

- in an ideal world one Trajectory would describe one particle Track, i.e. *ThreeVector x = traj( s ) ;* would give a correct description of the particle path as measured and give a reasonable extrapolation towards both ends
  - including proper description of momentum and cov. matrix
  - possibly one could interpolate between two or more helix track fits/parameterizations
    - -> mathematically unclear (to me)
- better to make full Trajectory persistent, e.g. by storing N points along fitted Trajectory
  - how many points needed ?
  - momenta and covariances needed for every point ?
  - file size unreasonably large ( DST have no hits for a reason !)
- implications on LCIO Track
  - already long discussions on current helix parameterization
  - -> probably a mid-to-long term project !?

# discussion

- even though making the Trajectory persistent is not so straight forward it would be possible to have a transient implementation based on points (hits) during tracking and PFA
  - -> is this needed / usefull ?
- the proposed interface is probably too simplistic, e.g. it needs to be extended by errors !
  - possibly every quantity could come with its error matrix
  - -> need proper class design depending on core vector and matrix implementation (CLHEP/MathMore/...)
  - should the computation of the error be optional ?
    - if so, is the default to compute or not compute the error ?

# Conclusion

- as originally proposed the Trajectory class should make the design of Track reconstruction software easier and make the results more correct
  - lots of issues (previous slides)
  - implications and usability unclear
- however currently we are working on PFA
  - need to extrapolate Tracks into calorimeter
  - need to extrapolate Tracks towards Vertices
- Proposal: introduce a Trajectory interface as presented and implement it through a simple Helix class and take it from there !
  - -> this will make code more extensible for future improvements and elaborations

**your input is welcome !!**