Reconstruction Notes

CAVEAT: Basically, I'm just using these pages as a notepad. The idea is to make the doxygen generated commets reflect the stuff here eventually

Overiew

The AcdRecon does a few things.

1) make MIP calibrated quantities from the digis (AcdHit)
2) calculate the distance between hit tiles & ribbons & track extrapolations -> (AcdTkrHitPoca)
3) calculate the intersection point of track extrapolations w/ the ACD (AcdTkrIntersection)
4) caculate the distance to the closest relevent gap in the ACD for track extrapolations (AcdTkrGapPoca & AcdCornerDoca)
5) extract quantities for the merit tuple. (Basically all the other stuff in Event::AcdRecon class)

Algorithm

This is just a list of the order & nesting the various functions are called in.

AcdReconAlg::reconstruct(const Event::AcdDigiCol& digiCol)

Is the main function. It takes the collection of AcdDigis from the event as input.

This function:
1) builds the set of AcdHits (AcdPha2MipTool::makeAcdHits())
2) calculates all the geometrical quantities for each track and the event vertex (trackDistances(), vertexDistances(), AcdTrkIntersectTool::exitsLAT(), AcdPocaTool::tileDistances(), AcdPocaTool::ribbonDistances())
3) latches best values for storage to Merit Tuple (doca(), hitTileDist(), tileAcdDist(), hitRibbonDist())
4) extrapolates track to the ACD (extrapolateTrack())
5) puts output on the TDS

AcdPha2MipTool::makeAcdHits(const Event::AcdDigiCol& digiCol, Event::AcdHitCol& hits, AcdRecon::AcdHitMap& hitMap)

Converts all the digis to calibrated AcdHits.

Normally all digis are converted to hits.

Depending on if the hit was read out in the high range or the low range different conversions are applied.
The low range uses a linear conversion:

mips = (PHA - pedestal) / mip_peak_PHA

The high range uses a form that is linear for low values, but saturates for high values:

mips = ((PHA - pedetsal) * saturation * slope) / (saturation + ((PHA - pedestal) * slope))

if ((PHA - pedestal) * slope << saturation) this goes to:
mips = (PHA - pedetsal) * slope

if ((PHA - pedestal) * slope >> saturation) this goes to:
mip = saturation

Both of the conversion functions live in the AcdUtil/AcdCalibFuncs. They are
mipEquivalent_lowRange
mipEquivalent_highRange

AcdReconAlg::trackDistances(...)

Does all the geometrical computations for each track.

Those calculations are:

1) finding the point where the track exits the nominal ACD in both directions (AcdTrkIntersectTool::exitsLAT())
2) find the docas to the various hit Acd tiles and ribbons (AcdPocaTool::tileDistances(), AcdPocaTool::ribbonDistances())

AcdPocaTool::tileDistances(...)

This gets the various types of doca & active distance calculations for tiles

AcdRecon::pointPoca()	-> doca to center of tile
AcdRecon::tilePlane()	-> point that track crosses tile plane & 2D active distance
AcdRecon::tileEdgePoca() or	-> doca to closest edge (track inside tile)
AcdRecon::tileEdgeCornerPoca()	-> doca to closest edge or corner (track outside tile)

AcdPocaTool::ribbonDistances(...)

This get the various types of doca & active distance calculations for ribbons

AcdRecon::ribbonPlane()	-> point that track crosses ribbon plane
AcdRecon::ribbonPoca()	-> doca to ray defined by ribbon

AcdReconAlg::doca(...)

Latches the best (smallest) values of doca to tile center for all tracks

AcdReconAlg::hitTileDist(...)

Latches the best (largest) values of 2D active distance to tiles for all tracks

AcdReconAlg::tileActDist(...)

Latches the best (largest) values of "tileActiveDistance" (2D inside tile, 3D outside tile) for all tracks

AcdReconAlg::hitRibbonDist(...)

Latches the best (largest) value of 2D active distance to ribbons for all tracks

AcdReconAlg::extrapolateTrack(...)

Uses GEANT propagator to take track parameters out tointersection and POCAs.   Calls:

AcdTkrIntersecttTool:makeIntersections()	-> uses GEANT propagator to caluclate intersection w/ ACD elements
AcdRecon::projectErrorAtPoca()	-> propagates error matrix to POCA

AcdTkrIntersectTool::makeIntersections()

Uses GEANT propagator to caluclate intersection w/ ACD elements.   If the struck element has not been hit, does the POCA calculations

AcdRecon::ribbonPlane() and AcdRecon::ribbonPoca() or
AcdRecon::tilePlane() and AcdRecon::tileEdgePoca()

AcdRecon::projectToPlane()

Gets the poca to the tile holes (holePoca())

Decides where closest gap is, then calls

gapPocaRibbon( )	-> if the track actually hits a ribbon that isn't in an overlap area, uses 3D poca to ribbon center
gapPocaTile( )	-> if the track hits a tile, uses -1 * 3D poca to tile edge
fallbackToNominal()	-> if the track doesn't hit any GEANT element, uses 2D "active distance" (5. - fabs(x-x0)) to gap

AcdReconAlg:: calcCornerDoca(...)

Calculates the DOCA to the Rays defined by the side edges of the ACD. Signed to reflect the directionality of the gaps.

Geometrical Functions

These functions are defined in AcdRecon/AcdReconFuncs.h and implemented in src/AcdReconFuncs.cxx

pointPoca(const Track& track, const Point& point, arcLength, doca, Point& poca)

Gets the point of closest approach between a track projection and a space point.

Inputs:

track	-> the track projectection data
point	-> the point in question

Outputs:

arcLength	-> distance along the track where the poca occurs => poca = track.m_point + arcLength * track.m_dir
doca	-> distance of clostest approach == | point - poca |
poca	-> the point of closest approach

crossesPlane(const Track& track, const Point& plane, int face, arcLength, Point& hitPoint)

Gets the point where a track projection crosses a plane.   This assumes that the plane is oriented along a cartiesen axis

Inputs:
  track -> the track projectection data
  point -> point at the center of the plane
  face   -> enum which defines the orientation/ side of the LAT the plane is on (top=0, -Y, -X, +Y, +X, bottom)

Outputs:
  arcLength -> distance along the track where the plane is crossed occurs => hitPoint = track.m_point + arcLength * track.m_dir
  hitPoint   -> the point where the track projection crosses the plane

crossesPlane(const Track& track, const Transform& trans, arcLength, Point& hitPoint)

Gets the point where a track projection crosses a plane.   This does not assume that the plane is oriented along a cartiesen axis

Inputs:
  track -> the track projectection data
  trans -> transformation for global coordinates to the tile coordinates

Outputs:
  arcLength -> distance along the track where the plane is crossed occurs => hitPoint = track.m_point + arcLength * track.m_dir
  hitPoint   -> the point where the track projection crosses the plane

rayDoca(const Track& track, const Point& p1, const Point& p2, RayDoca& rayDoca, double& edgeLen)

Gets the point where a track comes closest to the ray from point p1 to point p2.

Inputs:
  track -> the track projection data
  p1, p2 -> intial and final points of the ray, usally two corners of a tile or two end of a ribbon

Outputs:
  rayDoca -> object with intesection data
  edgeLen -> length along the edge at which the POCA occurs, we check this to make sure the poca occurs between the two ends

rayDoca_withCorner(const Track& track, const Ray& ray,

arcLength, rayLength, dist, Point& x, Vector& v)

Get the point where a track comes closest to a ray. Handles the ends of the ray correctly.

Inputs:
  track -> the track projection data
  ray   -> the ray in question

Outputs:
  arcLength -> length along track where POCA occurs
  rayLength ->   length along ray where POCA occurs
  dist   ->   DOCA
  x   ->   point along track there POCA occurs, (ie, POCA of track to ray)
  v   ->   vector from x to closest point on ray.   x-v must be on the ray

tilePlane(const Track& track, const Tile& tile, PocaData &data)

Gets the point where a track projection crosses a tile

Inputs:
  track -> the track projectection data
  tile   -> the geomertical information about the tile

Outputs:
  data.arcLength_plane -> distance along the track where the plane is crossed occurs => hitPoint = track.m_point + arcLength * track.m_dir
  data.activeX   -> position of the crossing point relative to the edge of the active area (>0 is in active area)
  data.activeY
  data.active2D -> the larger of activeX and activeY
  data.hitsPlane -> the point where the track projection crosses the plane in global coords
  data.inPlane   -> the point where the track projection crosses the plane in local coords (+x , +y, -x, -y edges in local frame)
  data.volume   -> which volume of the tile (0 = main, 1 = bent piece)

tilePlaneActiveDistance(cons Tile& tile, iVol, const Point& globalPoint, Point& localPoint, activeX, activeY)

Gets the active distance of the intersection.

Inputs:
  tile   -> the geomertical information about the tile
  iVol   -> which volume of the tile (0 = main, 1 = bent piece)
  globalPoint -> intersection point in global coords

Outputs:
  localPoint -> intersection point in local coords (+x , +y, -x, -y edges in local frame)
  activeX, activeY -> active distances in X and Y

tileEdgePoca(const Track& track, const Tile& tile,arcLength, dist, Point& poca, Vector& vector, int& region)

Gets the point where a track projection (that goes inside a tile) comes closest one of the edges of the tile

Inputs:
    track -> the track projectection data
    tile   -> the geomertical informatio about the tile

 Outputs:
    arcLength -> distance along the track where the plane is crossed occurs => hitPoint = track.m_point + arcLength * track.m_dir
    dist   -> the distance of closest approach between the track and the tile edge (in 3D)
    poca   -> the point of closest approach along the track to the tile edge
    vector -> the vector from the poca to the closest point on the tile edge
    region -> a code to show which edge of the tile was considered (+y,+x,-y,-x)

tileEdgeCornerPoca(const Track& track, const Tile& tile,arcLength, dist, Point& poca, Vector& vector, int& region)

Gets the point where a track projection (that goes outside a tile) comes closest one of the edges or corners of the tile

Inputs:
    track -> the track projectection data
    tile   -> the geomertical informatio about the tile

 Outputs:
    arcLength -> distance along the track where the plane is crossed occurs => hitPoint = track.m_point + arcLength * track.m_dir
    dist   -> the distance of closest approach between the track and the tile edge (in 3D)
    poca   -> the point of closest approach along the track to the tile edge
    vector -> the vector from the poca to the closest point on the tile edge
    region -> a code to show which edge of the tile was considered (y,+x,~~+y, -x edges, ++, +-~~, --, -+ corners)

ribbonPlane(const Track& track, const Ribbon& ribbon, PocaData& data)

Gets the point where a track projection crosses a plane.   This assumes that the plane is oriented along a cartiesen axis

Inputs:
    track -> the track projectection data
    ribbon -> the geomertical informatio about the tile

 Outputs:
    data.arcLengthPlane -> distance along the track where the plane is crossed occurs => hitPoint = track.m_point + arcLength * track.m_dir
    data.active2D   -> the distance of closest approach between the track and the ribbon
    data.hitsPlane   -> the point where the track projection crosses the plane
   data.volume   -> which segment of the ribbon

ribbonPoca(const Track& track, const Ribbon& ribbon, arcLength, ribbonLength, dist, Point& poca, Vector& vector, int& region)

Gets the point where a track projection crosses a plane.   This assumes that the plane is oriented along a cartiesen axis

Inputs:
    track -> the track projectection data
    ribbon -> the geomertical informatio about the tile

 Outputs:
    arcLength -> distance along the track where the plane is crossed occurs => hitPoint = track.m_point + arcLength * track.m_dir
   ribbonLength -> distance along the ribbon where the POCA occurs, 0 is center of ribbon
    dist   -> the distance of closest approach between the track and the ribbon
    poca   -> the point of closest approach along the track to the ribbon
    vector -> the vector from the poca to the closest point on the ribbon
    region -> a code to show which edge of the ribbon was considered (+,- in local coords)

Track Projection Functions

These functions are also defined int AcdRecon/AcdReconFuncs.h and implemented in src/AcdReconFuncs.cxx

errorAtXPlane(delta, const TkrTrackParams& track, HepMatrix& covAtPlane)

  errorAtXPlane(delta, const TkrTrackParams& track, HepMatrix& covAtPlane)
  errorAtXPlane(delta, const TkrTrackParams& track, HepMatrix& covAtPlane)

Projects the covarience martix onto a plane This assumes that the plane is oriented along a cartiesen axis

Inputs:
  delta -> normal distance from end of track to plane
  track -> the track parameters (esp. the cov. martix)

Outputs:
  covAtPlane -> the covarience matrix projected to the plane, expressed in local coords (XX, XY   /   YX, YY)

projectErrorAtPoca(const TrackData& track, const TkrTrackParams& trackParams, const Point& poca, const Vector& pocaVector, pocaError)

Projects the covarience martix along the vector between the POCA and the edge of the tile of ribbon

Inputs:
  track, trackParams -> the track data & track parameters (esp. the cov. martix)
  poca -> the point of closest approach to the tile or ribbon edge
  pocaVector -> the vector from the POCA to the closest edge

Outputs:
  pocaError -> the projection of the covareince matrix along the pocaVector

Internal Data structures

These structures are defined in AcdRecon/AcdReconStruct.h

They are only for passing around information with the AcdRecon code.   The structures that export code to the TDS and ROOT are defined in Event/Recon/AcdRecon

PocaData

this stores eveything we might want to know about where a track goes relative to a tile or ribbon

```
// stuff about the Tile or Ribbon
idents::AcdId m_id;            // The AcdId of the hit element

// stuff about the DOCA to the center of the tile or ribbon
double m_arcLengthCenter; // Length along the track to the POCA to the center of the tile
double m_docaCenter;        // The distance of closest aproach to the center of the tile
Point m_pocaCenter;           // The POCA to the center of the tile

// stuff about the point the track projection crosses the tile or ribbon plane
double m_arcLengthPlane;  // Length along the track to the plane of the detector
double m_activeX;             // 2D active distance in local X and Y ( For the sides X is horizontal, Y is vertical (aka Z) )
double m_activeY
double m_active2D;           // The distance of closest aproach to the relevent edge in 2D
Point m_inPlane;                // 3D point that track crosses detector plane
double m_localX;               // The local coordinates.   For the sides X is horizontal, Y is vertical (aka Z)
double m_localY;
double m_localCovXX;        // The local covarience terms.  For the sides X is horizontal, Y is vertical (aka Z)
double m_localCovXY;
double m_localCovYY;
double m_cosTheta;           // angle between track and plane normal
double m_path;                 // pathlength of track in the active material

// stuff about the POCA between the track projection and the closest edge or corner of the tile or ribbon
double m_arcLength;         // Length along the track to the poca
double m_active3D;          // The distance of closest aproach to the relevent edge in 3D
double m_active3DErr;      // The error on distance of closest aproach to the relevent edge in 3D
Point m_poca;                  // Point of closest approach
Vector m_pocaVector;        // Vector from Track to POCA

// stuff about where the POCA occurs relative to the tile or ribbon
int m_region;                  // One of the enums in "??"
```

TrackData

```
HepPoint3D   m_point;          // the start (or end) point of the track
HepVector3D m_dir;            // the direction of the track
double          m_energy;      // the energy of the track at the start point
int                m_index;        // the index number of this track
bool             m_upward;      // which side of track
```

ExitData

```
int    m_face;               // 0:top 1:-X 2:-Y 3:+X 4:+Y 5:bottom
double m_arcLength;   // Length along the track to the m_x
Point  m_x;                  // Intersection Point
```