```
/*
Glo Position -5.94995 -10753.9 -1806.62 glb (8.65128,-10750.6,-1806.62)
Glo Position    24.05 -10753.9 -1907.62 glb (9.30191,-10752.7,-1907.62)    : 24.
05 ??????
Glo Position -5.94995 -10753.9 -2008.62 glb (8.81592,-10753.9,-2008.62)
Glo Position -5.94995 -10753.9 -2109.62 glb (5.67749,-10753.8,-2109.62)
Glo Position -5.94995 -10753.9 -2210.62 glb (2.4078,-10753.2,-2210.62)
Glo Position -5.94995 -10753.9 -2311.62 glb (-1.45657,-10751.1,-2311.62)
Glo Position -5.94995 -10753.9 -2412.62 glb (-2.25651,-10752.5,-2412.62)
Glo Position -5.94995 -10753.9 -2513.62 glb (-3.02203,-10755.8,-2513.62)
Glo Position -5.94995 -10753.9 -2614.62 glb (-5.49412,-10760.2,-2614.62)
Glo Position -5.94995 -10753.9 -2715.62 glb (-8.77315,-10764.7,-2715.62)


pAnalysis;0 0  x -0.00594995 y -10.7689 zposs -2.69562
pAnalysis;1 0  x -0.00594995 y -10.754 zposs -2.59462
pAnalysis;2 0  x -0.00594995 y -10.754 zposs -2.49362
pAnalysis;3 0  x -0.00594995 y -10.754 zposs -2.39262
pAnalysis;4 0  x -0.00594995 y -10.754 zposs -2.29162
pAnalysis;5 0  x -0.00594995 y -10.754 zposs -2.19062
pAnalysis;6 0  x -0.00594995 y -10.754 zposs -2.08962
pAnalysis;7 0  x 0.00905005 y -10.754 zposs -1.98862
pAnalysis;8 0  x 0.00905005 y -10.754 zposs -1.88762
pAnalysis;9 0  x 0.00905005 y -10.754 zposs -1.78662


 */

#include <cmath>
#include "TMath.h"
#include <cassert>
#include "TSpline.h"
#include "TVector3.h"
#include <sys/time.h>
#include "vect_manager.h"
#include "SwimParticle.h"
#include "InoTrackFitAlg.h"
// #include "G4SystemOfUnits.hh"
//#include "MultiSimAnalysis.hh"

//#include "G4Material.hh"
//#include "G4ParticleDefinition.hh"
//#include "G4MuonPlus.hh"
//#include "G4MuonMinus.hh"
//#include "G4VEmFluctuationModel.hh"
//#include "G4EnergyLossForExtrapolator.hh"

#include <string>
#include <math.h>
#include <vector>
using std::vector;
#include <fstream>
#include <cstdlib>
#include "InoTrack.h"
#include <TRandom3.h>
#include "InoCluster.h"
#include "SwimSwimmer.h"
#include <Math/ProbFunc.h>
//#include "Interpolator.h"
#include "InoTrackSegment.h"
//#include "Math/Interpolator.h"
#define MINLAYER 3

InoTrackFitAlg::InoTrackFitAlg() {
   cout<<" InoTrackFitAlg::InoTrackFitAlg() " <<endl;
  pAnalysis      = MultiSimAnalysis::AnPointer;
  pFieldMap      = micalFieldPropagator::FdPointer;//GM

  icalGeometry= (InoGeometry_Manager::APointer)->icalGeometry;
```



```
  localmat       = new TGeoMaterial("Fe",55.845,26,7.874);

  //g4Edisp              = new G4EnergyLossForExtrapolator(1);
  PoissonRn      = new TRandom3(0);
  IcalRange      = new InoMuRange();

  debug_fit = true;
  inoTrackCand_pointer = new InoTrackCand_Manager();

  micalDetectorParameterDef* paradef = micalDetectorParameterDef::AnPointer;
  StripXWidth = (1/m)*paradef->GetXStrwd();
  StripYWidth = (1/m)*paradef->GetYStrwd();
  nLayer      = paradef->GetnLayer();
  cout<<"nLayer"<<nLayer<<endl;
  LayerThickness = (1/m)*2*(paradef->GetParlay(2)+paradef->GetParirlay(2));
  cout<<"LayerThickness"<<LayerThickness<<endl;
  ShiftInX = (1/m)*paradef->GetShiftInX();
  ShiftInY = (1/m)*paradef->GetShiftInY();
  ShiftInZ =paradef->GetShiftInZ(0);
  // cout<<"ShiftinXYZ  "<<ShiftInX<<"  " <<ShiftInY<<"  "<<ShiftInZ<<endl;
  for (int ij=0; ij<3; ij++) {
    pargasxyz[ij] = (1/cm)*paradef->GetPargas(ij);

  }
  cout<<doubleLa<<endl<<endl;
  for (unsigned ijk=0; ijk<doubleLa; ijk++) {
    //  ZPosLayer[ijk] = (1/m)*(-paradef->GetParino(2) + 2*(paradef->GetParhcoil
(2)+paradef->GetParcoilsupport(2)) + 2*(ijk+1)*paradef->GetParirlay(2) + (2*ijk+
1)*(paradef->GetParlay(2))) + ShiftInZ;

    ZPosLayer[ijk] = (1/m)*(paradef->GetRPCLayerPosZ(ijk)+ShiftInZ+paradef->GetI
NOroomPos(2)+ paradef->GetStackPosInRoom(2));

    cout<< "ZPosLayer "<< ijk <<" "<<1000* ZPosLayer[ijk]<<endl;
  }
  //

  IcalX = (1/m)*( (3)*(paradef->GetParino(0)) + 200);
  IcalY = (1/m)*paradef->GetParino(1);

  CorrTimeError = pAnalysis->GetCorrTimeError();
  UnCorrTimeError = pAnalysis->GetUnCorrTimeError();
  TimeError = pow((pow(CorrTimeError,2.) + pow(UnCorrTimeError,2.)),0.5);

  // cout<<"CorrTimeError = "<<CorrTimeError<<endl;
  // cout<<"UnCorrTimeError = "<<UnCorrTimeError<<endl;
  // cout<<"TimeError = "<<TimeError<<endl;

  UseGeoSwimmer = 0;
  if(pAnalysis->isXtermOut==1) {
    cout <<"\n InoTrackFitAlg::InoTrackFitAlg() "<<StripXWidth<<" "<<StripYWidth<<" "<<Layer
Thickness<<endl;
  }

  inoHit_pointer = InoHit_Manager::APointer;
  InoCluster_pointer = InoCluster_Manager::APointer;
}

InoTrackFitAlg::~InoTrackFitAlg() {
  //GMA need to clear after evergy events;
  for (unsigned int ij=0; ij< inoTrackCand_pointer->InoTrackCand_list.size(); ij
++) {
    if (inoTrackCand_pointer->InoTrackCand_list[ij]) {
      delete inoTrackCand_pointer->InoTrackCand_list[ij];
      inoTrackCand_pointer->InoTrackCand_list[ij]=0;
    }
  }
  inoTrackCand_pointer->InoTrackCand_list.clear();
  if (inoTrackCand_pointer) {
```

```
    delete inoTrackCand_pointer;
    inoTrackCand_pointer=0;
  }
  delete PoissonRn;
  delete IcalRange;
}
void InoTrackFitAlg::RunAlg( ) { // (AlgConfig & ac,  CandHandle &ch, CandContex
t &cx)
   cout<<"InoTrackFitAlg::RunAlg( ) {..."<<endl;
  InoTrack_Manager *ptrackCollection = InoTrack_Manager::APointer;
  //if(debug_fit)cout <<"-----------------------InoTrackFitAlg::RunAlg( ) "<<ptra
ckCollection->InoTrack_list.size()<<"-----------------------"<<endl;
 LOOP:

  inoTrackCand_pointer->InoTrackCand_list.clear();

  double MinFiltData[100][6];double MaxFiltData[100][6]; //GMA14
  int Ndt = ptrackCollection->InoTrack_list.size();
  if (Ndt>100) Ndt=100;

  if (ptrackCollection) {
    for (unsigned int itrk=0; itrk<ptrackCollection->InoTrack_list.size(); itrk+
+) {
      // Initialisations
      for (int ij=1; ij<2; ij++) {
        fFinderTrack = ptrackCollection->InoTrack_list[itrk];
        pAnalysis->nhits_finder[itrk]=fFinderTrack->ClustsInTrack.size();
//asm

        double FinderPathLength = 0.0;
        double FinderDistance= 0.0;

        fMT = false;

        ZIncreasesWithTime = DirectionFromFinderHits(ptrackCollection->InoTrack_
list[itrk], FinderPathLength, FinderDistance);

        cout<<"ZIncreasesWithTime" <<ZIncreasesWithTime<<endl;
        // ZIncreasesWithTime= DirectionFromFinderHitsOldFunc(ptrackCollection->
InoTrack_list[itrk], FinderPathLength, FinderDistance);
        cout<<"Track No. = "<<itrk<<endl;

        int nhits1 = ptrackCollection->InoTrack_list[itrk]->GetEntries();
        for(int ixj = 0; ixj < nhits1; ixj++) {
          cout<<"Xpos "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[ixj
]->GetXPos()<<", YPos "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[ixj]
->GetYPos()<<", ZPos = "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[ixj
]->GetZPos()<<", ZLay = "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[ix
j]->GetZPlane()<<endl;
          // ", Time = "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[i
xj]->GetTime()<<" "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[ixj]->
GetBegTime()<<" "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[ixj]->Ge
tEndTime()<<" "<<ptrackCollection->InoTrack_list[itrk]->ClustsInTrack[ixj]->GetH
itEntries()<<endl;
        }

        // if(!ZIncreasesWithTime) {
        //   pAnalysis->timeAsciiOutput<<"TrackNo. = "<<itrk<<endl;
        //   int nhits1 = ptrackCollection->InoTrack_list[itrk]->GetEntries();
        //   for(int ixj = 0; ixj < nhits1; ixj++) {
        //     pAnalysis->timeAsciiOutput<<"ZPos = "<<ptrackCollection->InoTrack
_list[itrk]->ClustsInTrack[ixj]->GetZPos()<<", ZLay = "<<ptrackCollection->InoTr
ack_list[itrk]->ClustsInTrack[ixj]->GetZPlane()<<", Time = "<<ptrackCollection->
InoTrack_list[itrk]->ClustsInTrack[ixj]->GetTime()<<endl;
        //   }
        // }


        InoTrack *trk = ptrackCollection->InoTrack_list[itrk];
```



```
        int nhits       = trk->GetEntries();
        double VtxX     = trk->ClustsInTrack[0]->GetXPos();
        double VtxY     = trk->ClustsInTrack[0]->GetYPos();
        double EndX     = trk->ClustsInTrack[nhits-1]->GetXPos();
        double EndY     = trk->ClustsInTrack[nhits-1]->GetYPos();

        //      if (fabs(fmod(VtxX,2.0))<0.1 || fabs(fmod(VtxX,2.0))>1.9 || fabs
(fmod(VtxY,2.0))<0.1 || fabs(fmod(VtxY,2.0))>1.9) {fMT = true;}
        // || fabs(fmod(EndX,2.0))<0.12 || fabs(fmod(EndX,2.0))>1.88 || fabs(fmo
d(EndY,2.0))<0.12 || fabs(fmod(EndY,2.0))>1.88

        //if (fabs(VtxY)<6.0 && ((FinderPathLength-FinderDistance)/FinderPathLen
gth)<0.01)                                      // Very small curvature at highe
r B => larger momenta
        fMT = true;

        if (ZIncreasesWithTime) {
          if (MinPlane>100 && ((FinderPathLength-FinderDistance)/FinderPathLengt
h)<0.01 && nhits<30) fMT = true;
        } else {
          if (MaxPlane <50 && ((FinderPathLength-FinderDistance)/FinderPathLengt
h)<0.01 && nhits<30) fMT = true;
        }

        fTrackCand = new InoTrackCand(ptrackCollection->InoTrack_list[itrk], ZIn
creasesWithTime); //VALGRIND

        if(debug_fit) {cout <<"trksize "<<ij << " "<<ptrackCollection->InoTrack_lis
t[itrk]->GetEntries()<<" "<<fTrackCand->GetClusterEntries()<<endl;}

        fTrackCand->SetFitType((ZIncreasesWithTime) ? 1 : 0);

        SaveData=false;
        SwimThroughShower=false;
        PassTrack=true;

        MaxPlane=       -20;
        MinPlane=       nLayer;
        OtLStrip=       0;
        //?????????????????????? Are these variables doing anything? We can remo
ve them if necessary
        DeltaZ=-99;
        DeltaPlane=-99;
        ShowerEntryPlane=-99;

        NIter=0;TotalNSwimFail=0; NumFinderStrips=0;
        x_k[5]=0;
        x_k_minus[5]=0;
        //?????????????????????? Are these variables doing anything? We can remo
ve them if necessary

        for (unsigned int jk=0; jk<5; ++jk) {
          x_k[jk]=0;
          x_k_minus[jk]=0;
          EndState[jk]  =0;
          prevstate[jk]=0;
          prevpredn[jk]=0;
          H_k[0][jk]=0;
          H_k[1][jk]=0;
          K_k[jk][0]=0;
          K_k[jk][1]=0;

          VtxCov[jk]=-999; EndCov[jk]=-999;
          for (unsigned int kl=0; kl<5; ++kl) {
            C_k[jk][kl]=0;
            C_k_minus[jk][kl]=0;
            C_k_intermediate[jk][kl]=0;
            F_k[jk][kl]=0;
            F_k_minus[jk][kl]=0;
```

```
            Q_k[jk][kl]=0;
            Q_k_minus[jk][kl]=0;
            Identity[jk][kl]=0;
          }
        }
        Identity[0][0]=1; Identity[1][1]=1; Identity[2][2]=1; Identity[3][3]=1;
Identity[4][4]=1;

        // Set initial parameters
        //-------------------------------------------------------
        x_k_minus[0]=fTrackCand->GetVtxU();
cout<<"Used X "<<x_k_minus[0]<<endl;
        x_k_minus[1]=fTrackCand->GetVtxV();
       cout<<"Used Y "<<x_k_minus[1]<<endl;

        if(fTrackCand->GetVtxDirCosZ()!=0.) {
          x_k_minus[2]=fTrackCand->GetVtxDirCosU()/fTrackCand->GetVtxDirCosZ();/
/cout<<"Used tx "<<x_k_minus[2]<<endl;
          x_k_minus[3]=fTrackCand->GetVtxDirCosV()/fTrackCand->GetVtxDirCosZ();/
/cout<<"Used ty "<<x_k_minus[3]<<endl;
        } else {
          x_k_minus[2]=fTrackCand->GetVtxDirCosU(); //This scenario will not app
ear, but keep it for unforseen case
          x_k_minus[3]=fTrackCand->GetVtxDirCosV();
        }

        x_k_minus[4]=0.0;

        StateIter[0]=0.0;
        StateIter[1]=0.0;
        StateIter[2]=0.0;
        StateIter[3]=0.0;
        StateIter[4]=0.0;

        //x_k_minus[5]=0.0;
        //x_k4_biased=0;

        xxin = x_k_minus[0];
        yyin = x_k_minus[1];
        txin = x_k_minus[2];
        tyin = x_k_minus[3];

        B_in = 0.0000000000;
        ds                      = 0.0;
        ChiSquare       = 0.0;
        GPL                     = 0.0;
        RNG                     = 0.0;
        I                       = 0.0;
        BetheBloch      = 0.0;
        MagicRatio      = 0.0;
        nHit            = ptrackCollection->InoTrack_list[itrk]->GetEntries();

        // Run the high level methods
        InitialFramework();

        RunTheFitter();

        bool check1;bool check2;
        for (int jk = 0; jk<6; jk++) {
          check1 = std::isnan(MinPlaneData[jk]);
          check2 = std::isnan(MinPlaneData[jk]);
          if (check1 == true || check2 == true)
            PassTrack=false;
          if (PassTrack == false)
            break;
          MinFiltData[itrk][jk] = MinPlaneData[jk];
          MaxFiltData[itrk][jk] = MaxPlaneData[jk];
        }
        for (unsigned int jk=0; jk<nLayer; ++jk)        {
```



```
          for (unsigned int kl=0; kl<FilteredData[jk].size(); kl++) {
            cout <<"FilteredData[jk].size() "<<jk<<" "<<kl<<" "<<FilteredData[jk].size()<
<endl;
            cout<<"FilteredData "<<FilteredData[jk][0].x_k0<<" "<<FilteredData[jk][0
].x_k1<<" "<<FilteredData[jk][0].x_k2<<" "<<FilteredData[jk][0].x_k3<<endl;
          }
        }


        if (pAnalysis->ihist < pAnalysis->nhistmx-1 && ij==1 && pAnalysis->isVis
Out>=2) {
          for (unsigned int jk=0; jk<nLayer; ++jk)      {
            for (unsigned int kl=0; kl<FilteredData[jk].size(); kl++) {
              // cout <<"FilteredData[jk].size() "<<jk<<" "<<kl<<" "<<FilteredDa
ta[jk].size()<<endl;
              if (InitTrkClustData[jk].size()>0)        {
                pAnalysis->gens_list[5][pAnalysis->ihist]->Fill(FilteredData[jk]
[kl].x_k0,
                                                                FilteredData[jk]
[kl].x_k1,
                                                                ZPosLayer[jk]+0.
05); //InitTrkClustData[jk][0].csh->GetZPos()+0.05);
                cout<<"pAnalysis->gens_list[5][pAnalysis->ihst]->Fill();"<<endl;

                vectGr  tmpgr;
                tmpgr.x = FilteredData[jk][kl].x_k0;
                tmpgr.y = FilteredData[jk][kl].x_k1;
                tmpgr.z = ZPosLayer[jk]+0.05; //SlcClustData[jk][0].csh->GetZPos
()+0.05;
                tmpgr.dx = 0.0;
                tmpgr.dy = 0.0;
                tmpgr.dz = 0.0;
                // pAnalysis->fitr_vect.push_back(tmpgr);
                if (pAnalysis->isVisOut==3)
                  pAnalysis->gens_vect[5].push_back(tmpgr);
              }
            }
          }
        }

        if (/*Ndt==1 && */fTrackCand->GetNDOF()>0 && fTrackCand->GetNDOF()<1000)
 {
          //cout<<"Kolahal"<<endl;
          inoTrackCand_pointer->InoTrackCand_list.push_back(fTrackCand);
        } else {
          fTrackCand=0;
        }
        //a     cout <<" fittrer "<< inoTrackCand_pointer->InoTrackCand_list.siz
e()<<endl;

        for (unsigned int jk=0; jk<doubleLa; ++jk) {
          InitTrkClustData[jk].clear();
          SlcClustData[jk].clear();
          TrkClustsData[jk].clear();
          FilteredData[jk].clear();
        }
      }
      //ij=1
    }

    // bool TEM;
    // Ndt = ptrackCollection->InoTrack_list.size();
    // cout<<"Total no. of Track segments: "<<Ndt<<endl;
    // if (Ndt>1 && PassTrack == true)  // {
    //   double TargetZ; bool alpha;
    //   ///////////////////////////////////////////////////////
    //   for (int jk=Ndt-1; jk>=0; jk--) {
    //  if (jk>0) {
    //    if (MinFiltData[jk-1][5] > MinFiltData[jk][5]) {
```

```
    //      double Tr1[6] = {0};
    //      double Tr2[6] = {0};
    //      // cout<<"alpha->true"<<endl;
    //      Tr1[0] = MaxFiltData[jk][0];        Tr1[1] = MaxFiltData[jk][1];
Tr1[2] = MaxFiltData[jk][2];
    //      Tr1[3] = MaxFiltData[jk][3];        Tr1[4] = MaxFiltData[jk][4];
Tr1[5] = MaxFiltData[jk][5];
    //      // sometime sign of the qbyP at the end of the tracjk is wrong. So,
we borrow the sign from the start of the next
    //      //try this
    //      Tr1[4] = fabs(MaxFiltData[jk][4]) * (MaxFiltData[jk-1][4]/fabs(MaxFi
ltData[jk-1][4]));
    //      TargetZ= MinFiltData[jk-1][5];

    //      TrackElementMerging(Tr1, TargetZ, Tr2);
    //      alpha = true;
    //    } else {
    //      double Tr1[6] = {0};
    //      double Tr2[6] = {0};
    //      // cout<<"alpha->false"<<endl;
    //      Tr1[0] = MaxFiltData[jk-1][0];      Tr1[1] = MaxFiltData[jk-1][1];
Tr1[2] = MaxFiltData[jk-1][2];
    //      Tr1[3] = MaxFiltData[jk-1][3];      Tr1[4] = MaxFiltData[jk-1][4];
Tr1[5] = MaxFiltData[jk-1][5];
    //      // sometime sign of the qbyP at the end of the track is wrong. So, w
e borrow the sign from the start of the next
    //      //try this
    //      Tr1[4] = fabs(MaxFiltData[jk-1][4]) * (MaxFiltData[jk][4]/fabs(MaxFi
ltData[jk][4]));
    //      TargetZ= MinFiltData[jk][5];

    //      TrackElementMerging(Tr1, TargetZ, Tr2);
    //      alpha =false;
    //    }
    //    //cout<<"Before adding, no. of clusters in 0-th track is"<<ptrackColle
ction->InoTrack_list[jk-1]->ClustsInTrack.size()<<endl;
    //    if (alpha == true) {
    //      ptrackCollection->InoTrack_list[jk]->AddTrack(ptrackCollection->InoT
rack_list[jk-1]);
    //    } else if (alpha ==false) {
    //      ptrackCollection->InoTrack_list[jk-1]->AddTrack(ptrackCollection->In
oTrack_list[jk]);
    //    }
    //    vector <InoTrack*>::iterator it;
    //    it = ptrackCollection->InoTrack_list.begin();

    //    if (alpha == true) {
    //      ptrackCollection->InoTrack_list.erase(it+jk-1);
    //    } else if (alpha ==false) {
    //      ptrackCollection->InoTrack_list.erase(it+jk);
    //    }

    //    if (ptrackCollection->InoTrack_list.size()==1) {
    //      TEM = true;
    //    }
    //  }
    //   }

    //   if (TEM == true) {
    //  goto LOOP;
    //   }
    // }



    //--------------------------------------------------------------------------
----------------------------------------------
    //asm: this part of the code was inserted to change the order of the tracks
```



```
in trackCand vector.
    //Now the vectors are placed so that the segment close to the vertex part wi
ll taken placs as zeroth element of the vector.
    //Apoorva:      this part of the code was inserted to change the order of th
e tracks in trackCand vector.
    // Now the vectors are placed in decreasing order of their length
    vector <InoTrackCand*>::iterator itt2;
    if(inoTrackCand_pointer->InoTrackCand_list.size()>1) {
      InoTrackCand* tempTrk;
      InoTrack* tmpfinder;
      for(unsigned int ij=0; ij<inoTrackCand_pointer->InoTrackCand_list.size();i
j++) {
        for(unsigned int jk=ij+1; jk<inoTrackCand_pointer->InoTrackCand_list.siz
e();jk++) {
          //cout<< inoTrackCand_pointer->InoTrackCand_list.size() << "::"<<endl;
          if(inoTrackCand_pointer->InoTrackCand_list[ij]->GetMomentumdS() < inoT
rackCand_pointer->InoTrackCand_list[jk]->GetMomentumdS()) {
            cout<<"Yo Yo...."<<endl;
            tempTrk = inoTrackCand_pointer->InoTrackCand_list[jk];
            tmpfinder = ptrackCollection->InoTrack_list[jk];
            inoTrackCand_pointer->InoTrackCand_list[jk] = inoTrackCand_pointer->
InoTrackCand_list[ij];
            ptrackCollection->InoTrack_list[jk] = ptrackCollection->InoTrack_lis
t[ij];
            inoTrackCand_pointer->InoTrackCand_list[ij] = tempTrk;
            ptrackCollection->InoTrack_list[ij] = tmpfinder;
          }
        }
      }

      bool TEM = false;
      bool combTracks = true;
      if(combTracks) {
        for(unsigned int ij=0; ij<inoTrackCand_pointer->InoTrackCand_list.size()
;ij++) {
          for(unsigned int jk=ij+1; jk<inoTrackCand_pointer->InoTrackCand_list.s
ize();jk++) {
            int zplaneEndTrk1 = inoTrackCand_pointer->InoTrackCand_list[ij]->Get
EndPlane();
            int zplaneVtxTrk1 = inoTrackCand_pointer->InoTrackCand_list[ij]->Get
VtxPlane();
            int zplaneEndTrk2 = inoTrackCand_pointer->InoTrackCand_list[jk]->Get
EndPlane();
            int zplaneVtxTrk2 = inoTrackCand_pointer->InoTrackCand_list[jk]->Get
VtxPlane();
            int iRPCmodVtx1 = inoTrackCand_pointer->InoTrackCand_list[ij]->GetVt
xRPCmod();
            int iRPCmodEnd1 = inoTrackCand_pointer->InoTrackCand_list[ij]->GetEn
dRPCmod();
            int iRPCmodVtx2 = inoTrackCand_pointer->InoTrackCand_list[jk]->GetVt
xRPCmod();
            int iRPCmodEnd2 = inoTrackCand_pointer->InoTrackCand_list[jk]->GetEn
dRPCmod();
            int nInCHend1 = iRPCmodEnd1%8;
            iRPCmodEnd1>>=3;
            int nInMOend1 = iRPCmodEnd1%8;
            int nInCHvtx1 = iRPCmodVtx1%8;
            iRPCmodVtx1>>=3;
            int nInMOvtx1 = iRPCmodVtx1%8;
            int nInCHend2 = iRPCmodEnd2%8;
            iRPCmodEnd2>>=3;
            int nInMOend2 = iRPCmodEnd2%8;
            int nInCHvtx2 = iRPCmodVtx2%8;
            iRPCmodVtx2>>=3;
            int nInMOvtx2 = iRPCmodVtx2%8;
            bool VtxEndMatchZ = false;
            bool VtxEndMatchCHMO = false;
            if(inoTrackCand_pointer->InoTrackCand_list[ij]->GetFitType()==1) {
              if(zplaneEndTrk1>zplaneVtxTrk1) {
```

```
              if((zplaneEndTrk2>zplaneEndTrk1) && (zplaneVtxTrk2>zplaneEndTrk1
)) {
                VtxEndMatchZ = true;
                if((abs(nInCHend1-nInCHend2) == 1) || (abs(nInCHend1-nInCHvtx2
)==1) || (abs(nInMOend1-nInMOend2) == 1) || (abs(nInMOend1-nInMOvtx2)==1)) {
                  VtxEndMatchCHMO = true;
                } else {
                  VtxEndMatchCHMO = false;
                }
              } else {
                VtxEndMatchZ = false;
              }
              if((zplaneEndTrk2<zplaneVtxTrk1) && (zplaneVtxTrk2<zplaneVtxTrk1
)) {
                VtxEndMatchZ = true;
                if((abs(nInCHvtx1-nInCHend2) == 1) || (abs(nInCHvtx1-nInCHvtx2
)==1) || (abs(nInMOvtx1-nInMOend2) == 1) || (abs(nInMOvtx1-nInMOvtx2)==1)) {
                  VtxEndMatchCHMO = true;
                } else {
                  VtxEndMatchCHMO = false;
                }
              } else {
                VtxEndMatchZ = false;
              }
            }
          } else {
            if(zplaneEndTrk1<zplaneVtxTrk1) {
              if((zplaneEndTrk2<zplaneEndTrk1) && (zplaneVtxTrk2<zplaneEndTrk1
)) {
                VtxEndMatchZ = true;
                if((abs(nInCHend1-nInCHend2) == 1) || (abs(nInCHend1-nInCHvtx2
)==1) || (abs(nInMOend1-nInMOend2) == 1) || (abs(nInMOend1-nInMOvtx2)==1)) {
                  VtxEndMatchCHMO = true;
                } else {
                  VtxEndMatchCHMO = false;
                }
              } else {
                VtxEndMatchZ = false;
              }
              if((zplaneEndTrk2>zplaneVtxTrk1) && (zplaneVtxTrk2>zplaneVtxTrk1
)) {
                VtxEndMatchZ = true;
                if((abs(nInCHvtx1-nInCHend2) == 1) || (abs(nInCHvtx1-nInCHvtx2
)==1) || (abs(nInMOvtx1-nInMOend2) == 1) || (abs(nInMOvtx1-nInMOvtx2)==1)) {
                  VtxEndMatchCHMO = true;
                } else {
                  VtxEndMatchCHMO = false;
                }
              } else {
                VtxEndMatchZ = false;
              }
            }
          } // if(inoTrackCand_pointer->InoTrackCand_list[0]->GetFitType()==1)
 {

          if(VtxEndMatchCHMO && VtxEndMatchZ) {
            ptrackCollection->InoTrack_list[ij]->AddTrack(ptrackCollection->In
oTrack_list[jk]);
            vector <InoTrack*>::iterator itt3;
            itt3 = ptrackCollection->InoTrack_list.begin();
            ptrackCollection->InoTrack_list.erase(itt3+jk);
            goto LOOP;
          }
        } // for(unsigned int jk=ij+1; jk<inoTrackCand_pointer->InoTrackCand_l
ist.size();jk++) {
      } // for(unsigned int ij=0; ij<inoTrackCand_pointer->InoTrackCand_list.s
ize();ij++) {
    } // if(combTracks) {
  }
```



```


    //------------------------------------------------------------------------
-----------------------------------------------
    //asm: this part of the code was inserted to change the order of the tracks
 in trackCand vector.
    //Now the vectors are placed so that the segment close to the vertex part wi
ll taken placs as zeroth element of the vector.

    bool tmpnui = false;
    if(tmpnui) {
      cout<<"LOL...."<<endl;
      vector <InoTrackCand*>::iterator itt1;
      if(inoTrackCand_pointer->InoTrackCand_list.size()>1) {
        InoTrackCand* tempTrk;

        for(unsigned int jk=1; jk<inoTrackCand_pointer->InoTrackCand_list.size()
;jk++) {
          //cout<< inoTrackCand_pointer->InoTrackCand_list.size() << "::"<<endl;

          if(abs(inoTrackCand_pointer->InoTrackCand_list[0]->GetVtxZ()-inoTrackC
and_pointer->InoTrackCand_list[jk]->GetVtxZ())>1) {

            if(inoTrackCand_pointer->InoTrackCand_list[0]->GetFitType()==1 &&ino
TrackCand_pointer->InoTrackCand_list[jk]->GetFitType()==1  ) {
              if(inoTrackCand_pointer->InoTrackCand_list[0]->GetVtxZ()>inoTrackC
and_pointer->InoTrackCand_list[jk]->GetVtxZ()) {
                tempTrk = inoTrackCand_pointer->InoTrackCand_list[jk];
                itt1=inoTrackCand_pointer->InoTrackCand_list.begin();
                inoTrackCand_pointer->InoTrackCand_list.erase(itt1+jk);
                inoTrackCand_pointer->InoTrackCand_list.insert(itt1,tempTrk);

              }
            } else if ( inoTrackCand_pointer->InoTrackCand_list[0]->GetFitType()
==0 && inoTrackCand_pointer->InoTrackCand_list[jk]->GetFitType()==0) {
              if(inoTrackCand_pointer->InoTrackCand_list[0]->GetVtxZ()<inoTrackC
and_pointer->InoTrackCand_list[jk]->GetVtxZ()) {

                tempTrk = inoTrackCand_pointer->InoTrackCand_list[jk];
                itt1=inoTrackCand_pointer->InoTrackCand_list.begin();
                inoTrackCand_pointer->InoTrackCand_list.erase(itt1+jk);
                inoTrackCand_pointer->InoTrackCand_list.insert(itt1,tempTrk);
              }
            }
          }
        }
      }
    }
    //------------------------------------------------------------------------
-----------------------------------------------
  }

  // cout<<"...} InoTrackFitAlg::RunAlg()"<<endl;
}

void InoTrackFitAlg::TrackElementMerging(double *Tr1, double TargetZ, double *Tr
2) {
  double Bx = 0; double By = 0;
  double Position[3]={0};       Position[0]=Tr1[0]; Position[1]=Tr1[1]; Position
[2]=Tr1[5];

  //    micalFieldPropagator *pFieldMap;
  pFieldMap = micalFieldPropagator::FdPointer;

  while(Position[2]>=Tr1[5] && Position[2] < TargetZ) {
    // Scale the Position array for calling Magnetic Field
    Position[0] *= 1000;        Position[1] *= 1000;    Position[2] *= 1000;
```

```
    pFieldMap->ElectroMagneticField(Position,Bx,By,1);
    Bx *= 1000; By *= 1000;



    // ReScale the Position array after calling Magnetic Field
    Position[0] /= 1000;        Position[1] /= 1000;    Position[2] /= 1000;


    // Take small steps
    double dz = 0.002;

    double x; double y; double tx; double ty; double qbyP; double z;
    x = Tr1[0]; tx= Tr1[2];     qbyP= Tr1[4];
    y = Tr1[1]; ty= Tr1[3];     z       = Tr1[5];

    double pos[3]={0.0};
    double dir[3]={0.0};

    //int signp = (ZIncreasesWithTime != GoForward) ? -1 : 1;

    double dxdz = tx;
    double dydz = ty;
    double dsdz = pow((1.+pow(tx,2)+pow(ty,2)),0.5);

    pos[0] = 1.e2*x;
    pos[1] = 1.e2*y;
    pos[2] = 1.e2*z;

    dir[0] = dxdz/dsdz;
    dir[1] = dydz/dsdz;
    dir[2] = 1.0/dsdz;

    double state[5] ={x,y,tx,ty,qbyP};

    icalGeometry->InitTrack(pos, dir);
    localmat = icalGeometry->GetCurrentVolume()->GetMaterial();
    double Eloss = 0;
    Eloss = GetEnergyLoss(state, dz, xi, T_max, I, localmat);
    //cout<<"Eloss "<<Eloss<<endl;

    double P = 0; double E = 0; double amu = 0.1056; double P_ex;       double E
_ex;
    {
      P = fabs(1/qbyP);
        //cout<<"PSC(iron) P = "<<P<<endl;
      E = sqrt(pow(P,2) + pow(amu,2));
//cout<<"PSC(iron) E = "<<E<<endl;
      E_ex= E - Eloss;

      if ((E_ex-amu)>0) {
        P_ex= sqrt(pow(E_ex,2) - pow(amu,2));
      } else {
        P_ex= P;
      }
    }

    double kappa        = 0.299792458;
    double T            = sqrt(1+pow(tx,2)+pow(ty,2));
    //    double T2              = pow(T,2);
    double h            = kappa*qbyP*T;
    double Rx           = Bx*dz*h;
    //    double Rxx = 0.5 * Bx * Bx * pow(dz,2);
    //    double Sxx = (1/6)*Bx*Bx*pow(dz,3);
    double Ry           = By*dz*h;
    //    double Rxy = 0.5 * Bx * By * pow(dz,2);
    //    double Sxy = (1/6)*Bx*By*pow(dz,3);
    double Sx           = 0.5*Bx*pow(dz,2)*h;
    //    double Ryx = 0.5 * By * Bx * pow(dz,2);
```



```
    //    double Syx = (1/6)*By*Bx*pow(dz,3);
    double Sy           = 0.5*By*pow(dz,2)*h;
    //    double Ryy = 0.5 * By * By * pow(dz,2);
    //    double Syy = (1/6)*By*By*pow(dz,3);

    Tr1[0] = x + tx * dz + tx * ty * Sx - (pow(tx,2)+1) * Sy;// + h*h * (tx*(3*t
y*ty+1)*Sxx - ty*(3*tx*tx+1)*Sxy -ty*(3*tx*tx+1)*Syx + tx*(3*ty*ty+3)*Syy);
    if (fabs(qbyP)>4.0) Tr1[0] = x + tx * dz;

    Tr1[1] = y + ty * dz + (pow(ty,2)+1) * Sx - tx * ty * Sy;// + h*h * (ty*(3*t
y*ty+3)*Sxx - tx*(3*ty*ty+1)*Sxy -tx*(3*ty*ty+1)*Syx + ty*(3*tx*tx+1)*Syy);
    if (fabs(qbyP)>4.0) Tr1[1] = y + ty * dz;

    Tr1[2] = tx + tx * ty * Rx - (pow(tx,2)+1) * Ry;// + h*h * (tx*(3*ty*ty+1)*R
xx - ty*(3*tx*tx+1)*Rxy -ty*(3*tx*tx+1)*Ryx + tx*(3*ty*ty+3)*Ryy);
    if (fabs(qbyP)>4.0) Tr1[2] = tx;

    Tr1[3] = ty + (pow(ty,2)+1) * Rx - tx * ty * Ry;// + h*h * (ty*(3*ty*ty+3)*R
xx - tx*(3*ty*ty+1)*Rxy -tx*(3*ty*ty+1)*Ryx + ty*(3*tx*tx+1)*Ryy);
    if (fabs(qbyP)>4.0) Tr1[3] = ty;

    Tr1[4] = Tr1[4] * (P/P_ex);

    Tr1[5] = Position[2] + dz;

    Position[2] = Position[2] + dz;

    //cout<<"Tr1X "<<Tr1[0]<<"     "<<"Tr1Y "<<Tr1[1]<<"     "<<"Tr1Z "<<Tr1[5]<
<"     "<<"Tr1Tx "<<Tr1[2]<<"     "<<"Tr1Ty"<<Tr1[3]<<endl;
  }
  for (int ij = 0; ij<6; ij++) {
    Tr2[ij] = Tr1[ij];
  }
  //cout<<"Extrapolated X "<<Tr2[0]<<"     "<<"Extrapolated Y "<<Tr2[1]<<endl;
}

void InoTrackFitAlg::InitialFramework() { //const CandSliceHandle* slice,CandCon
text &cx)
  if(debug_fit) { cout<<" InoTrackFitAlg::InitialFramework() " <<endl;}
  // Store InoHit and make the strips accessible by plane number
  double MisalignmentError=0.0; //2.5e-5;

  //  double MisalignmentError=1e-8; //1e-6; //1e-4; //4e-4; //1e-8; //4e-6;  //
 GMA need number from INO : Squared error for misalignment of strips
  //  double strXwd = StripXWidth; // 0.0196;  //GMA use common variable, this i
s in metre (NOT IN CM)
  //  double XposErrorSq = pow(strXwd/pow(12.,0.5),2.);
  //  double strYwd = StripYWidth; //0.0196;  //GMA use common variable, this is
 in metre (NOT IN CM)
  //  double YposErrorSq = pow(strYwd/pow(12.,0.5),2.);

  int SlicePlane;

  // a cout <<"inside InitialFramework "<<endl;
  // Store all clusters

  //a  cout <<"1size "<< InoCluster_pointer->InoCluster_list.size()<<endl;
  for (unsigned ij=0; ij<InoCluster_pointer->InoCluster_list.size(); ij++) {
    SlicePlane=InoCluster_pointer->InoCluster_list[ij]->GetZPlane();
    ClustStruct temp;
    temp.csh=InoCluster_pointer->InoCluster_list[ij];
    SlcClustData[SlicePlane].push_back(temp);
  }

  int TrackPlane;

  //  Store all track clusters found,
  for (unsigned ij=0; ij<fFinderTrack->ClustsInTrack.size(); ij++) {
    SlicePlane=fFinderTrack->ClustsInTrack[ij]->GetZPlane();
```

```
    ClustStruct temp;
    temp.csh=fFinderTrack->ClustsInTrack[ij];

    TrackPlane=temp.csh->GetZPlane();

    TrkDataStruct tempdata;
    tempdata.numInList = ij;
    tempdata.cltime =temp.csh->GetTime();

    tempdata.ZPos=ZPosLayer[TrackPlane];
    tempdata.PlaneView = temp.csh->GetView();

    tempdata.XPos=temp.csh->GetXPos();
    tempdata.XPosErrSq = pow(temp.csh->GetXPosErr(),2.0) + MisalignmentError ; /
/ + XposErrorSq; // pow(temp.csh->GetXPosErr(),2.0);

    tempdata.YPos=temp.csh->GetYPos();
    tempdata.YPosErrSq = pow(temp.csh->GetYPosErr(),2.0) + MisalignmentError; //
  + YposErrorSq; // pow(temp.csh->GetYPosErr(),2.0);

    tempdata.Straight = fFinderTrack->ClustsInTrack[ij]->GetStraight();
    int ishift = (fFinderTrack->ClustsInTrack[ij]->GetStraight()) ? 0 : shiftLa;
    TrkClustsData[TrackPlane+ishift].push_back(tempdata);
    InitTrkClustData[SlicePlane+ishift].push_back(temp);

    if (ishift >0) {
      temp.csh->SetStraight(false);
    } else {
      temp.csh->SetStraight(true);
    }

    // Identify ends of initial track
    if (TrackPlane>MaxPlane) {MaxPlane=TrackPlane;}
    if (TrackPlane<MinPlane) {MinPlane=TrackPlane;}
  }
  //a cout <<"Exiting InitialFramework "<<MinPlane<<" "<<MaxPlane<<endl;
}


// void InoTrackFitAlg::ShowerStrips() {
//   //a  cout <<"InoTrackFitAlg : ShowerStrips, Look for large vertex shower" <
< endl;
//   // It is not being used currently (even not with implimenation of Geometry)
: Kolahal (Nov5,2013)
//   // Initialisations
//   int Increment; int NumberOfHits;
//   int Plane; int NewPlane;

//   int VtxShwWindow=8;
//   int HitsForShw=4;
//   double PEThreshold=0.00001; //GMA  .1;

//   if(ZIncreasesWithTime==true) {
//     Plane=MinPlane; Increment=1;
//   } else {
//     Plane=MaxPlane; Increment=-1;
//   }
//   NewPlane=Plane;

//   // Identify any vertex showers
//   while(abs(Plane-NewPlane)<=VtxShwWindow && NewPlane>=MinPlane && NewPlane<=
MaxPlane) {
//     if(SlcClustData[NewPlane].size()>0) {
//       NumberOfHits=0;
//       // Set the number of hits on a plane required for the plane to be ident
ified as 'in the
//       // shower'. We account for the gradient of the track, with the factor o
f 0.25 representing
//       // the approximate ratio of strip thickness to strip width.
```



```
//       if(FilteredData[NewPlane].size()>0) {
//      //GMA Need optimisation
//      //GMA 07/02/2009 Excluding layer, which does not have any hit points,
//      //but in track extrapolation, it is stored.
//      if (FilteredData[NewPlane][0].x_k4 !=0.0) {
//        if(SlcClustData[NewPlane][0].csh->GetView()==2) {
//          //GMA what is this view
//          HitsForShw=max(min(7,int( 4+(0.25*fabs(FilteredData[NewPlane][0].x_k
2)) )),min(7,int( 4+(0.25*fabs(FilteredData[NewPlane][0].x_k3)) )));
//        } else if (SlcClustData[NewPlane][0].csh->GetView()==0) {
//          HitsForShw=min(7,int( 4+(0.25*fabs(FilteredData[NewPlane][0].x_k2))
));
//        } else {
//          HitsForShw=min(7,int( 4+(0.25*fabs(FilteredData[NewPlane][0].x_k3))
));
//        }
//      }
//       } else {
//      HitsForShw=4;
//       }

//       // Count number of strips on plane with greater than 2PEs
//       for(unsigned int ij=0; ij<SlcClustData[NewPlane].size(); ++ij) {
//      if(SlcClustData[NewPlane][ij].csh->GetPulse()>PEThreshold) {NumberOfHits
++;}
//       }

//       // If a vertex shower is found, note that we should use the Swimmer
//       // to find the most likely track strips inside the shower
//       if(NumberOfHits>=HitsForShw) {ShowerEntryPlane=NewPlane; SwimThroughSho
wer=true; break;}
//       NewPlane+=Increment;
//     } else {
//       NewPlane+=Increment;
//     }
//   }
//   // Find the plane at which the 'clean' section of track enters the shower
//   if(SwimThroughShower==true) {
//     NewPlane=ShowerEntryPlane+Increment;
//     int PlanesSinceLastHit=0;
//     int PlaneWindow=4;

//     while(PlanesSinceLastHit<PlaneWindow && NewPlane>=MinPlane && NewPlane<=M
axPlane) {
//       if(SlcClustData[NewPlane].size()>0) {
//      NumberOfHits=0;

//      // Account for gradient of track, as before
//      if(FilteredData[NewPlane].size()>0) {
//        // GMA 07/02/2009 Excluding layer, which does not have any hit points,
//        // but in track extrapolation, it is stored.
//        if (FilteredData[NewPlane][0].x_k4 !=0.0) {
//          if(SlcClustData[NewPlane][0].csh->GetView()==2) {
//            HitsForShw=max(min(7,int(4+(0.25*fabs(FilteredData[NewPlane][0].x_
k2)) )),min(7,int(4+(0.25*fabs(FilteredData[NewPlane][0].x_k3)) )));
//          } else if (SlcClustData[NewPlane][0].csh->GetView()==0) {
//            HitsForShw= min(7,int(4+(0.25*fabs(FilteredData[NewPlane][0].x_k2)
) ));
//          } else {
//            HitsForShw=min(7,int(4+(0.25*fabs(FilteredData[NewPlane][0].x_k3))
 ));
//          }
//        }
//      } else {
//        HitsForShw=4;
//      }

//      // Count number of strips on plane with greater than 2PEs
```

```
//       for(unsigned int ij=0; ij<SlcClustData[NewPlane].size(); ++ij) {
//         if(SlcClustData[NewPlane][ij].csh->GetPulse()>PEThreshold) {
//           NumberOfHits++;
//         }
//       }
//       if(NumberOfHits>=HitsForShw) {
//         ShowerEntryPlane=NewPlane; NewPlane+=Increment; PlanesSinceLastHit=0;
//       } else {
//         PlanesSinceLastHit++; NewPlane+=Increment;
//       }
//        } else {
//       PlanesSinceLastHit++; NewPlane+=Increment;
//        }
//      }
//   }
// }

void InoTrackFitAlg::RunTheFitter( ) {
  cout<< " InoTrackFitAlg::RunTheFitter()" <<endl;
  if(debug_fit) { cout<< " InoTrackFitAlg::RunTheFitter()" <<endl;}

  // cout <<"InoTrackFitAlg : RunTheFitter, Call methods in the appropriate orde
r" << endl;
  GetInitialCovarianceMatrix(true);

  const bool GoForward=true;
  const bool GoBackward=false;

  double StateVector[6] = {0.0};
  double Prediction[6]  = {0.0};

  // Control the iterations backwards and forwards
  // Detector::Detector_t detector = vldc->GetDetector();

  int niteration=5;
  double chisq_old=-100;
  int ndof_old = -100;
  int ndifchi = -1;
  //  double CSQ[5]={0.0};

  // Control iterations over a track for which ZIncreasesWithTime
  if(ZIncreasesWithTime==true) {
    cout<<"ZIncreasesWithTime"<<endl;

    //First iteration
    NIter++;

    //Vtx to End (Forwards)

    SaveData=true;
    StoreFilteredData(MinPlane);
    LastIteration=false;

    GoForwards(false);
    //ResetCovarianceMatrix();

    //cout <<"true============ResetCovarianceMatrix============================
"<<endl;

    // ShowerStrips() was not explicitly given here; it is explicitly givrn (alb
eit commented) for ZDecreasesWitTime. GMA did not use it at all.
    // ShowerStrips();
    if(SwimThroughShower==true) {
      RemoveTrkHitsInShw();
    }
    // SwimThroughShower conditionally set to true only within ShowerStrips() wh
ich is
    // never called. So, effectively, this command does nothing.
```



```
    for (unsigned int ij=0; ij<doubleLa; ++ij) {
      FilteredData[ij].clear();
    }

    StoreFilteredData(MaxPlane);

    //End to Vtx: Backwards

    GoBackwards(false);

    if(SwimThroughShower==true) {
      ShowerSwim();
    }
    // SwimThroughShower conditionally set to true only within ShowerStrips() wh
ich is
    // never called. So, effectively, this command does nothing.

    for(int ij= 0; ij<5; ij++){StateIter[ij] = x_k[ij];}
    //cout<<"C00 "<<C_k[0][0]<<"     C11 "<<C_k[1][1]<<"     C22 "<<C_k[2][2]<<"
     C33 "<<C_k[3][3]<<"     C44 "<<C_k[4][4]<<endl;
    ResetCovarianceMatrix();

    bool ClusterFound = true;

    // FindTheStrips(false);
    // cth,false);
    // asm: here we can set some check to decide when no cluster found

    if(ClusterFound==true) { //GMA 210625 Why this
      // cout<<"cluster found"<<endl;
      // Guard against finding no strips
      for(int nint=0; nint <= niteration; nint++) {
        double fi = 0.0;
        for(int ij = 0; ij<5; ij++) {
          if(NIter>2)
            fi = (x_k[ij]-StateIter[ij])/StateIter[ij];
          StateIter[ij] = x_k[ij];
          /*if (fabs(fi) > 0.1)
            cout<<fi<<" for ij= "<<ij<<" @ N "<<NIter<<endl;*/
        }

        double chisq_new = fTrackCand->GetChi2();
        int ndof_new = fTrackCand->GetNDOF();
        //cout<<chisq_old<<"     "<<chisq_new<<"     "<<ndifchi<<"     "<<NIter<
<endl;

        if (ndof_old == ndof_new && abs(chisq_old - chisq_new) < 0.01) {
          ndifchi=0;
        }
        //Valgrind comments : Conditional jump or move depends on uninitialised
value

        ndof_old = ndof_new;
        chisq_old = chisq_new;

        //GMA stop loop if there is no change in hit points
        //Keep in mind SaveData=true (Data from only last iteration is stored);

        NIter++;

        if((nint==niteration || ndifchi==0) && nint>1) {
          LastIteration = true;
        }
        if (nint>0) {
          GetFitData(MinPlane,MaxPlane);
        }
        // cout<<"GetFitData_new "<<"MinPlane = "<<MinPlane<<"   "<<"MaxPlane =
"<<MaxPlane<<endl;
        if (MinPlane > MaxPlane) {
```

```
          cout<<" PassTrack 3a " << MinPlane<< "<"<< MaxPlane<<endl;
          PassTrack=false;
          break;
        }
        //a cout <<"2true=====ResetCovarianceMatrix====="<<endl;

        SaveData=true;                          // Here Savedata was set to fals
e
        StoreFilteredData(MinPlane);// This command was absent. It is introduced
 to keep parity with GB

        fTrackCand->f2dS.clear();
        fTrackCand->f2Range.clear();

        GoForwards(true);

        //Abhijit's Work. ADB 2015/05/06
        //>>
        // Extrapolating upwards
        //Upward = 1 , Downward = 0
        // cout<<"NIter FCorPCForward "<<NIter<<endl;
        if (NIter == 4 ) {
          // cout<<"XXXXXXXXXXXXXXXXXX"<<endl;
          FCorPCForward = CheckFCPCUpOrDn(x_k, 1, MaxPlane, GoForward);
          // cout<<"1FCorPCForward "<<FCorPCForward<<endl;
          // cout<<"------------------------------"<<endl;
        }
        //<<
        //Abhijit's Work ADB 2015/05/06


        //ResetCovarianceMatrix();
        //Look on this

        //End back to vtx again
        for (unsigned int ij=0; ij<doubleLa; ++ij) {
          for (unsigned jk=0; jk<FilteredData[ij].size(); jk++) {
            if (FilteredData[ij][jk].x_k5==1) {
              FilteredData[ij].erase(FilteredData[ij].begin()+jk);
              jk--;
            }
          }
        }

        SaveData=true;
        StoreFilteredData(MaxPlane);

        fTrackCand->fdS.clear();
        fTrackCand->fRange.clear();

        GoBackwards(false);
        //Abhijit's Work. ADB 2015/05/06
        //>>
        // Extrapolating downwards
        //Upward = 1 , Downward = 0
        // cout<<"NIter FCorPCBackward "<<NIter<<endl;
        if (NIter == 4 ) {
          FCorPCBackward = CheckFCPCUpOrDn(x_k, 0, MinPlane, GoBackward);
          // cout<<"2FCorPCBackward "<<FCorPCBackward<<endl;
          // cout<<"XXXXXXXXXXXXXXXXXX"<<endl;
        }
        //<<
        //Abhijit's Work ADB 2015/05/06
        ResetCovarianceMatrix();

        if(nint==0)
          x_k4_biased= x_k[4];

        if ((nint == niteration || ndifchi==0) && nint>1) {
```



```
            //int nextplane=100;
            //bool GetPrediction=PredictedStateCov(StateVector, MinPlane, nextplan
e, GoBackward, Prediction, 0);
            bool ok1 = true;
            if(pAnalysis->isXtermOut==1) {
              if (ok1) {  //GetPrediction
                for (int ij=0; ij<6; ij++) {
                  cout <<ij<<" "<<StateVector[ij]<<" "<< Prediction[ij]<<endl;
                }
                cout <<" end "<< 1/StateVector[4]<<" "<<1/Prediction[4]<<endl;
              }
            }
            break;
          }
          // if (ndifchi==0) break;
        }
      } else { // clusterfound
        cout<<" PassTrack 3.1 " <<endl;
        PassTrack=false;
      }
    } else {
      cout<<" Control iterations over a track for which ZDecreasesWithTime"<<"NIter "<<NIter<<endl;
      // First iteration
      NIter++;

      // Vtx to End (Backwards)
      SaveData=true;
      StoreFilteredData(MaxPlane);
      LastIteration=false;

      GoBackwards(false);
      ResetCovarianceMatrix();

      // ShowerStrips();
      if(SwimThroughShower==true) {
        RemoveTrkHitsInShw();
      }
      // SwimThroughShower conditionally set to true only within ShowerStrips() wh
ich is
      // never called. So, effectively, this command does nothing.

      for (unsigned int ij=0; ij<doubleLa; ++ij) {
        FilteredData[ij].clear();
      }

      StoreFilteredData(MinPlane);

      // End to Vtx: Forwards
      GoForwards(false);

      if(SwimThroughShower==true) {
        ShowerSwim();
      }
      // SwimThroughShower conditionally set to true only within ShowerStrips() wh
ich is
      // never called. So, effectively, this command does nothing.

      ResetCovarianceMatrix();

      bool ClusterFound = true;
      // GMAA FindTheStrips(false); // cth,false);
      // bool ClusterFound = FindTheStrips(false); // cth,false);
      // Second iteration

      if(ClusterFound==true) {
        // Guard against finding no strips
        for(int nint=0;nint<=niteration;nint++) {
          double chisq_new = fTrackCand->GetChi2();
          int ndof_new = fTrackCand->GetNDOF();
```

```
        if (ndof_old == ndof_new && abs(chisq_old - chisq_new) < 0.01) {
          ndifchi=0;
        }
        ndof_old = ndof_new;
        chisq_old = chisq_new;

        NIter++;

        if((nint==niteration || ndifchi==0) && nint>1)
          LastIteration = true;

        if (nint>0)
          GetFitData(MinPlane,MaxPlane);

        if (MinPlane > MaxPlane) {
          cout<<" PassTrack 3.2 " << MinPlane<< "<"<< MaxPlane<<endl;
          PassTrack=false;
          break;
        }

        SaveData=true;                          // Here Savedata was set to fals
e
        StoreFilteredData(MaxPlane);// This command was absent. It is introduced
 to keep parity with GF

        fTrackCand->f2dS.clear();
        fTrackCand->f2Range.clear();

        // if (nint==0) { GoBackwards(true); } else { GoBackwards(false); }

        GoBackwards(true);
        //Abhijit's Work. ADB 2015/05/06
        //>>
        // Extrapolating downwards
        //Upward = 1 , Downward = 0
        // cout<<"NIter FCorPCBackward "<<NIter<<endl;
        if (NIter == 4 ) {
          // cout<<"XXXXXXXXXXXXXXXXXX"<<endl;
          FCorPCBackward = CheckFCPCUpOrDn(x_k, 0, MinPlane, GoForward);
          // cout<<"-------------------------------"<<endl;
          // cout<<"3FCorPCBackward "<<FCorPCBackward<<endl;
        }
        //<<
        //Abhijit's Work ADB 2015/05/06
        ResetCovarianceMatrix();

        // End to Vtx again
        for (unsigned int ij=0; ij<doubleLa; ++ij) {
          for (unsigned jk=0; jk<FilteredData[ij].size(); jk++) {
            if (FilteredData[ij][jk].x_k5==1) {
              FilteredData[ij].erase(FilteredData[ij].begin()+jk);
              jk--;
            }
          }
        }
        // if (TrkClustsData[i].size()>0) FilteredData[i].clear();}
        // if(nint==niteration || ndifchi==0 )

        SaveData=true;
        StoreFilteredData(MinPlane);

        fTrackCand->fdS.clear();
        fTrackCand->fRange.clear();

        GoForwards(false);
        //Abhijit's Work. ADB 2015/05/06
        //>>
        // Extrapolating upwards
        //Upward = 1 , Downward = 0
```



```
        // cout<<"NIter FCorPCForward "<<NIter<<endl;
        if (NIter == 4 ) {
          FCorPCForward = CheckFCPCUpOrDn(x_k, 1, MaxPlane, GoBackward);
          // cout<<"4FCorPCForward "<<FCorPCForward<<endl;
          // cout<<"XXXXXXXXXXXXXXXXXX"<<endl;
        }
        //<<
        //Abhijit's Work ADB 2015/05/06

        ResetCovarianceMatrix();

        if(nint==0) {
          x_k4_biased= x_k[4];
        }
        if ((nint == niteration || ndifchi==0 )&&nint>1) {

          //GMA Move this prediction to new code
          //      int nextplane=100;
          //bool GetPrediction=PredictedStateCov(StateVector, MaxPlane, nextplan
e, GoForward, Prediction, 1);
          bool ok2 = true;

          if(pAnalysis->isXtermOut==1) {
            if (ok2) {          //GetPrediction
              for (int ij=0; ij<6; ij++) {
                cout <<ij<<" "<<StateVector[ij]<<" "<< Prediction[ij]<<endl;
              }
              cout <<" endf "<< 1/StateVector[4]<<" "<<1/Prediction[4]<<endl;
            }
          }
          break;
        }
        // if (ndifchi==0) break;
      }
    } else {
      cout<<" PassTrack 3.3 " << endl;
      PassTrack=false;
    }
  }

  // checkfcorpc = 0;
  FCorPC = 0;
  // G4int fcpc_tmp123 =0;
  if(NIter==4) {

    FCorPC = (((FCorPCBackward<<4)&0x0fff0)|(FCorPCForward&0x0f));
    // cout<<"FCorPC = "<<FCorPC<<", checkfcorpc = "<<checkfcorpc<<endl;
    // cout<<" FCPC = "<<FCPC<<", FCorPC = "<<FCorPC<<", FCorPCForward = "<<FCor
PCForward<<", FCorPCBackward = "<<FCorPCBackward<<endl;
  }
  // Organise the output
  if(pAnalysis->isXtermOut==1) {
    for (int ij=MinPlane; ij<=MaxPlane; ij++) {
      for (unsigned jk=0; jk<FilteredData[ij].size(); jk++) {
        cout<<"iMax "<<ij<<" "<<jk<<" "<<FilteredData[ij][jk].x_k5<<" "<<FilteredD
ata[ij][jk].x_k0<<" "<<FilteredData[ij][jk].x_k1<<" "<<FilteredData[ij][jk].x_k2<
<" "<<FilteredData[ij][jk].x_k3<<" "<<1./(FilteredData[ij][jk].x_k4)<<" "<<Filter
edData[ij][jk].x_k5<<endl;
      }
    }
  }

  if(pAnalysis->isXtermOut==1) {
    if (MaxPlane >=0 && MaxPlane <int(doubleLa)) {
      cout<<endl;
      //cout<<"-----------------------------------------------------------------
--"<<endl;
      //cout<< " indx "<<" Pln no " <<"FilteredData.x_k5 "<<"x_k0 " << "x_k1 " <
< "x_k2 " << "x_k3 " << "1./x_k4 "<< "x_k5 "<<endl;
```

```
      cout<<endl;
      for (unsigned ij=0; ij<FilteredData[MaxPlane].size(); ij++) {
        cout<<"iMax "<<ij<<" "<<MaxPlane<<" "<<FilteredData[MaxPlane][ij].x_k5<<"
"<<FilteredData[MaxPlane][ij].x_k0<<" "<<FilteredData[MaxPlane][ij].x_k1<<" "<<Fi
lteredData[MaxPlane][ij].x_k2<<" "<<FilteredData[MaxPlane][ij].x_k3<<" "<<1./(Fil
teredData[MaxPlane][ij].x_k4)<<" "<<FilteredData[MaxPlane][ij].x_k5<<endl;
      }
    }
    if (MinPlane >=0 && MinPlane <int(doubleLa)) {
      for (unsigned ij=0; ij<FilteredData[MinPlane].size(); ij++) {
        cout<<"iMin "<<ij<<" "<<MinPlane<<" "<<FilteredData[MinPlane][ij].x_k5<<" "
<<FilteredData[MinPlane][ij].x_k0<<" "<<FilteredData[MinPlane][ij].x_k1<<" "<<Fil
teredData[MinPlane][ij].x_k2<<" "<<FilteredData[MinPlane][ij].x_k3<<" "<<1./(Filt
eredData[MinPlane][ij].x_k4)<<" "<<FilteredData[MinPlane][ij].x_k5<<endl;
      }
    }
  }

  double PI = 3.14159;
  double theta = 0.00;
  theta = (180/PI) * acos(1/sqrt(1+pow(x_k[2],2)+pow(x_k[3],2))); //GMA 210625 S
imilarly for phi ?
  if (ZIncreasesWithTime==false) {
    theta = (180/PI) * acos(-1.0/sqrt(1+pow(x_k[2],2)+pow(x_k[3],2)));
  }

  int n = 0;
  InoTrack_Manager *ptrackCollection = InoTrack_Manager::APointer;

  // If the fit was successful
  // ChiSquare  = 0; // ChiSquare;  //GMA14
  // GPL                         = 0.5 * GPL;
  // RNG                         = 0.5 * RNG;
  // In equations above, the factor '0.5' comes to account for the double counti
ng of forward and backward iteration
  //  cout<<"GPL "<<GPL<<"     "<<"RNG "<<RNG<<endl;

  // if(ptrackCollection->InoTrack_list.size() == 1 && x_k[4]!=0. && PassTrack==
true) {
  n = ptrackCollection->InoTrack_list[0]->ClustsInTrack.size();

  // cout<<"-----------------------------------------------------------------------
------------------------------------------------------"<<endl;
  // cout<<"Reconstructed P = "<<1/x_k[4]<<"  |  "<<"theta "<<theta<<"  |  L "<<
FinderPathLength<<"  |  MaxPlane "<<MaxPlane<<"  |  MinPlane "<<MinPlane<<"  |
#hits "<<n<<"  |  chisq/dof "<<ChiSquare/(2*n-5)<<endl;
  // cout<<"-----------------------------------------------------------------------
-----------------------------------------------------"<<endl;

  // cout<<"No. of hits for this track was "<<n<<"     "<<"MinPlane "<<MinPlane<
<"     "<<"MaxPlane "<<MaxPlane<<endl;
  //cout<<"ChiSquare/dof "<<ChiSquare/(2*n-5)<<endl;

  FillGapsInTrack();

  bool FinalClusterFound = true; // GMA
  if(FinalClusterFound==true) {
    int NumInUView = fTrackCand->GetNPlane(0);
    int NumInVView = fTrackCand->GetNPlane(1);

    //    cout <<"numview "<< NumInUView <<" "<<NumInVView<<" "<<fTrackCand->Get
ClusterEntries()<<endl;
    //if((ChiSquare/(2*n-5))<10 && (ChiSquare/(2*n-5))>0.01 &&(NumInUView>1 && N
umInVView>1))

    if (NumInUView>1 && NumInVView>1)   {
      SetTrackProperties(x_k); //cth);
    } else {
      PassTrack=false;
```



```
    }
  } else { // Otherwise fail the track at this final stage
    cout<<" PassTrack 6" << endl;
    PassTrack=false;
  }
  // }
  // If the fit has failed (e.g. q/p is zero and/or u, v are nonsense)
  if(x_k[4]==0. || PassTrack==false) {
  }

}

void InoTrackFitAlg::RemoveTrkHitsInShw() {
  // If the 'clean' section of track is large enough, remove the track finding
  // data for planes before the ShowerEntryPlane
  if(debug_fit)  { cout <<"InoTrackFitAlg : RemoveTrkHitsInShw, Discard track finding data in shower"
 << endl;}

  int NumTrackHitsLeft=0;

  if(ZIncreasesWithTime==true) {
    for(int ij=ShowerEntryPlane; ij<=MaxPlane; ++ij) {
      if(TrkClustsData[ij].size()>0) {NumTrackHitsLeft++;}
    }
  } else if(ZIncreasesWithTime==false) {
    for(int ij=MinPlane; ij<=ShowerEntryPlane; ++ij) {
      if(TrkClustsData[ij].size()>0) {NumTrackHitsLeft++;}
    }
  }

  // Carry out removal if there will be 6 or more strips left afterwards
  if(NumTrackHitsLeft>5) {
    if(ZIncreasesWithTime==true) {
      for(int ij=MinPlane; ij<=ShowerEntryPlane; ++ij) {TrkClustsData[ij].clear(
);}
    } else if(ZIncreasesWithTime==false) {
      for(int ij=ShowerEntryPlane; ij<=MaxPlane; ++ij) {TrkClustsData[ij].clear(
);    }
    }
  } else {  // Otherwise note that we should not run the ShowerSwim method
    cout <<"InoTrackFitAlg : RemoveTrkHitsInShw, not enough hits after removal. Must use all finder data." <
< endl;
    SwimThroughShower=false;
  }

  // Find the new max and min planes
  MaxPlane=-20; MinPlane=5000;
  for (int ij=0; ij<(int)nLayer; ++ij) {
    if(TrkClustsData[ij].size()>0) {
      if(ij>MaxPlane) {MaxPlane=ij;}
      if(ij<MinPlane) {MinPlane=ij;}
    }
  }
}

// void InoTrackFitAlg::ShowerSwim() {
//   // Method is called if we have a large shower near the track vertex
//   // The Swimmer is used to find the most likely track strip in the shower
//   // and this strip is added to the fit

//   if(debug_fit) {
//     cout<<" =========================================================="<<endl
;
//     cout <<"InoTrackFitAlg : ShowerSwim, improved track finding in shower" <<
 endl;
//     cout<<" =========================================================="<<endl
;
//   }
```

```
//   // Initialisations
//   int Plane; int NewPlane;
//   double StateVector[6]; double NState[6]; double x__minus[6];
//   bool GoForward; bool SwimBack;
//   int PlanesSinceLastHit=0;
//   int PlaneView;
//   int Increment;

//   double StripXDistance=0; double StripYDistance=0;
//   double MinXDistanceToStrip=999; double MinYDistanceToStrip=999.;
//   //  double StripXWidth=2.00e-2; double StripYWidth=2.00e-2;// 4.108e-2;

//   if(ZIncreasesWithTime==true) {
//     GoForward=false; Plane=MinPlane; Increment=-1;
//   } else {
//     GoForward=true; Plane=MaxPlane; Increment=1;
//   }

//   NewPlane=Plane+Increment;

//   // Continue until we reach a 4 plane window with no likely hit or we reach
//   // the end of the detector

//   while(PlanesSinceLastHit<4 && NewPlane>0 && NewPlane<=(int)nLayer-5) {
//     //145) { //GMA Put those number from database
//     if(SlcClustData[NewPlane].size()>0) {
//       PlaneView = SlcClustData[NewPlane][0].csh->GetView();
//       for(int ij=0; ij<6; ++ij) {
//         StateVector[ij]=x_k_minus[ij];
//       }

//       SwimBack=Swim(StateVector, NState, Plane, NewPlane, GoForward);

//       if(!SwimBack) {
//         break;
//       }
//       for(int ij=0; ij<6; ++ij) {
//         x_k[ij]=NState[ij];
//       }

//       // Find the closest strip (within a distance 'MinDistanceToStrip') and
//       // temporarily store CandStripHandle
//       // Results are very sensitive to value of MinDistanceToStrip

//       InoCluster* CurrentClust=0;

//       //GMA Original 0.0055, but do not have much clue about it
//       // Is it (0.01*gap/stripwidth) ? Then for INO it is 0.0426
//       //Was put 0.0852 also why donot remember now (30/01/08)

//       MinXDistanceToStrip=(1.5*StripXWidth)+ fabs(0.0055*x_k[2]); //Original
//       MinYDistanceToStrip=(1.5*StripYWidth)+ fabs(0.0055*x_k[3]);

//       for(unsigned int j=0; j<SlcClustData[NewPlane].size(); ++j) {
//         if (PlaneView%2==0) StripXDistance=fabs(SlcClustData[NewPlane][j].csh
->GetXPos()-x_k[0]);
//         if (PlaneView>0) StripYDistance=fabs(SlcClustData[NewPlane][j].csh->G
etYPos()-x_k[1]);

//         if(StripXDistance<MinXDistanceToStrip && StripYDistance<MinYDistanceT
oStrip) {
//           if (PlaneView%2==0) MinXDistanceToStrip=StripXDistance;
//           if (PlaneView>0)    MinYDistanceToStrip=StripYDistance;
//           CurrentClust=SlcClustData[NewPlane][j].csh;
//         }
//       }

//       // If we find a likely track strip, add it to the fit data and call the
 Kalman
```



```
//       // update equations before repeating process to find next track strips
in the shower
//       //      cout <<"CurrentClust "<<int(CurrentClust)<<" "<<NewPlane<<" "<<
SlcClustData[NewPlane].size()<<endl;
//       //      cout <<"CurrentClust  "<<NewPlane<<" "<<SlcClustData[NewPlane].
size()<<endl;
//       if(CurrentClust) {
//         ClustStruct temp;
//         temp.csh = CurrentClust;
//         InitTrkClustData[NewPlane].push_back(temp);

//         // Convert the strip to data required for Kalman fit
//         GetFitData(NewPlane,NewPlane);

//         // Carry out the Kalman fit
//         for (int ij=0; ij<2; ij++) {
//           for (int jk=0; jk<5; jk++) {
//             H_k[ij][jk]=0;
//           }
//         }
//         if (PlaneView%2==0) {
//           H_k[0][0]=1;
//         }
//         if (PlaneView>0) {
//           H_k[1][1]=1;
//         }
//         //     cout <<"InoTrackFitAlg.Showerswim : WARNING : PlaneView for hi
ts is not matching with 0/1/2"<<endl;


//         // bool CombiPropagatorOk=GetCombiPropagator(Plane,NewPlane,GoForward
);
//         bool CombiPropagatorOk = 1;
//         if(CombiPropagatorOk) {

//           //GetMultipleScattering(istate,dz,eloss);
//           //GetEnergyLoss(istate,dz);
//           //ExtrapolatedCovariance(ifmultiplescattering);
//           //cout<<"State Vector (1st): "<<StateVector[0]<<"  "<<StateVector[1
]<<"    "<<StateVector[2]<<"    "<<StateVector[3]<<"    "<<StateVector[4]<<"
"<<StateVector[5]<<"    "<<endl;
//           ds = 0.0;
//           drange = 0.0;
//           PredictedStateCov(x_k_minus,Plane,NewPlane, GoForward, x__minus, 0,
 &ds, &drange);
//           CalcKalmanGain(x__minus,NewPlane);
//           //UpdateStateVector(i,NewPlane,direction); //true);
//           KalmanFilterStateVector(x__minus, Plane, GoForward,x_k);
//           //UpdateStateVector(Plane,NewPlane,true);
//           UpdateCovMatrix(NewPlane);
//           MoveArrays(NewPlane,GoForward);


//           StoreFilteredData(NewPlane);

//           if(ZIncreasesWithTime) {
//             MinPlane=NewPlane;
//             Plane=MinPlane;
//           } else {
//             MaxPlane=NewPlane;
//             Plane=MaxPlane;
//           }
//           NewPlane=Plane+Increment;

//           PlanesSinceLastHit=0;
//         }
//       } else {
//         NewPlane+=Increment; PlanesSinceLastHit++;
//       }
```

```
//      } else {
//        NewPlane+=Increment; PlanesSinceLastHit++;
//      }
//    }
//    // Note that shower swim is complete
//    SwimThroughShower=false; //GMA why false ?
// }

void InoTrackFitAlg::GetFitData(int& Plane1, int& Plane2) {
  // Loop over the initial track strip data and create the final data for fittin
g
  if(debug_fit) { cout <<"InoTrackFitAlg : GetFitData "<<"Plane1 "<<Plane1<<" Plane2 "<<Plane
2<<endl;}

  // Initialsmislaiisations
  double MisalignmentError=0.0; //2.5e-5; //  double MisalignmentError=1e-8; //1
e-6; //1e-4; //4e-4; //1e-8; //4e-6;  // GMA need number from INO : Squared erro
r for misalignment of strips

  Plane1=10000;
  Plane2 = -20;
  fTrackCand->ClustsInTrack.clear();
  //cout<<"doubleLa = "<<doubleLa<<endl;
  for (unsigned int ij=0; ij<doubleLa; ++ij) {
    InitTrkClustData[ij].clear();
    TrkClustsData[ij].clear();
  }

  for (unsigned ijk=0; ijk<doubleLa; ijk++) {
    if (FilteredData[ijk].size()==0)
      continue;
    // cout <<"ijlayer "<<ijk<<" "<<FilteredData[ijk].size()<<endl;

    for (unsigned ij=0; ij<FilteredData[ijk].size(); ij++) {
      double x1 = FilteredData[ijk][ij].x_k0;
      double y1 = FilteredData[ijk][ij].x_k1;
      // cout<<"x1 = "<<x1<<"     "<<"y1 = "<<y1<<"     "<<"shiftLa = "<<shiftLa
<<endl;
      int jk = (ijk >=shiftLa) ? ijk-shiftLa : ijk;

      double dmn = 0.07; //Maximum 10 cm               asm: note this 270711
      int klx = -1;

      for(unsigned int kl=0; kl<SlcClustData[jk].size(); ++kl) {
        double dx = fabs(SlcClustData[jk][kl].csh->GetXPos()-x1); // /SlcClustDa
ta[jk][kl].csh->GetXPosErr();
        double dy = fabs(SlcClustData[jk][kl].csh->GetYPos()-y1); // /SlcClustDa
ta[jk][kl].csh->GetYPosErr();
        if (dmn > pow(dx*dx+dy*dy,0.5)) {
          dmn = pow(dx*dx+dy*dy,0.5);
          klx=kl;
        }
      }

      if (klx>=0) {
        fTrackCand->ClustsInTrack.push_back(SlcClustData[jk][klx].csh);
        // InitTrkClustData[jk].push_back(SlcClustData[jk][klx]);
        // const InoCluster* tempcls = SlcClustData[jk][klx].csh;

        int TrackPlane= SlcClustData[jk][klx].csh->GetZPlane();
        TrkDataStruct tempdata;
        tempdata.numInList = fTrackCand->GetClusterEntries()-1;

        tempdata.cltime =SlcClustData[jk][klx].csh->GetTime();
        tempdata.ZPos=ZPosLayer[TrackPlane];
        tempdata.PlaneView =  SlcClustData[jk][klx].csh->GetView();
        tempdata.XPos=  SlcClustData[jk][klx].csh->GetXPos();
        tempdata.XPosErrSq = pow(SlcClustData[jk][klx].csh->GetXPosErr(),2.0) +
MisalignmentError; //  + XposErrorSq; // pow(temp.csh->GetXPosErr(),2.0);
```



```
        tempdata.YPos= SlcClustData[jk][klx].csh->GetYPos();
        tempdata.YPosErrSq = pow(SlcClustData[jk][klx].csh->GetYPosErr(),2.0) +
MisalignmentError ; // + YposErrorSq; // pow(temp.csh->GetYPosErr(),2.0);

        //      cout <<"tmpdata "<< jk<<" "<< klx<<" "<<tempdata.cltime <<" "<<S
lcClustData[jk][klx].csh->GetTime()<<" "<<tempdata.XPos<<" "<<tempdata.YPos<<" "
<<tempdata.ZPos<< endl;

        tempdata.Straight = FilteredData[ijk][ij].x_k6;
        int ishift = (FilteredData[ijk][ij].x_k6) ? 0 : shiftLa;
        TrkClustsData[TrackPlane+ishift].push_back(tempdata);
        InitTrkClustData[jk+ishift].push_back(SlcClustData[jk][klx]);
        if (ishift >0) {
          SlcClustData[jk][klx].csh->SetStraight(false);
        } else {
          SlcClustData[jk][klx].csh->SetStraight(true);
        }

        // TrkClustsData[TrackPlane].push_back(tempdata);
        // cout <<"TrackPlane1 "<< TrackPlane<<" "<<Plane1<<" "<<Plane2<<endl;
        // cout<<"ishift = "<<ishift<<endl;
        if (TrackPlane>Plane2 && ishift==0) Plane2 = TrackPlane;
        if (TrackPlane<Plane1 && ishift==0) Plane1 = TrackPlane;
        // cout <<"TrackPlane2 "<< TrackPlane<<" "<<Plane1<<" "<<Plane2<<endl;
      } else {
        // FilteredData[ijk].clear();
        FilteredData[ijk].erase(FilteredData[ijk].begin()+ij);
        ij-- ;
      }
    }
  }
}

void InoTrackFitAlg::FillGapsInTrack() {
  // If there is no filtered data for a plane (between MinPlane and MaxPlane),
  // but this plane has hits in the slice, we interpolate from the nearest
  // state vectors
  //
  // As with all filtered data, the interpolated data will be compared to
  // strip positions in the FindTheStrips method
  if (debug_fit) { cout <<"InoTrackFitAlg :: FillGapsInTrack" << endl;}

  int CurrentPlane; int ForwardsPlane; int BackwardsPlane;
  int Plane; int NewPlane;  bool GoForward;
  double StateVector[6]; double Prediction[6]; bool GetPrediction;

  for (int ij=MinPlane; ij<=MaxPlane; ++ij) {
    if(SlcClustData[ij].size()>0) {
      if(FilteredData[ij].size()==0) {
        // Find nearest filtered state vectors (within two planes) and ZPos diff
erences
        // Forwards
        CurrentPlane=ij+1; ForwardsPlane=-99;
        while(CurrentPlane<=MaxPlane && CurrentPlane<=(ij+2)) {
          if(FilteredData[CurrentPlane].size()>0) {
            ForwardsPlane=CurrentPlane; break;
          } else {
            CurrentPlane++;
          }
        }

        // Backwards
        CurrentPlane=ij-1; BackwardsPlane=-99;

        while(CurrentPlane>=MinPlane && CurrentPlane>=(ij-2) ) {
          if(FilteredData[CurrentPlane].size()>0) {
            BackwardsPlane=CurrentPlane; break;
          } else {
            CurrentPlane--;
```

```
          }
        }

        // Find and store possible new filtered data, range and dS
        if(ForwardsPlane!=-99 && BackwardsPlane!=-99) {
          // Swimmer method
          GetPrediction=false;
          NewPlane=ij;
          if(ZIncreasesWithTime==true) {Plane=ForwardsPlane; GoForward=false;}
          else{Plane=BackwardsPlane; GoForward=true;}
          if(FilteredData[Plane].size()>0) {
            StateVector[0] = FilteredData[Plane][0].x_k0;
            StateVector[1] = FilteredData[Plane][0].x_k1;
            StateVector[2] = FilteredData[Plane][0].x_k2;
            StateVector[3] = FilteredData[Plane][0].x_k3;
            StateVector[4] = FilteredData[Plane][0].x_k4;
            StateVector[5] = (double)FilteredData[Plane][0].x_k5;
            GetPrediction=Swim(StateVector, Prediction, Plane, NewPlane, GoForwa
rd);

            if(GetPrediction==true) {
              // Store possible new state vector
              FiltDataStruct temp;
              temp.x_k0 = Prediction[0];
              temp.x_k1 = Prediction[1];
              temp.x_k2 = Prediction[2];
              temp.x_k3 = Prediction[3];
              temp.x_k4 = Prediction[4];
              temp.x_k5 = int(Prediction[5]);
              temp.x_k6 = true; //11Nov2009
              // FilteredData[ij].clear(); // 110809
              FilteredData[ij].push_back(temp);
            }
          }
        }
      }
    }
  }
}

bool InoTrackFitAlg::Swim(double* StateVector, double* Output, const int Plane,
                          const int NewPlane,const bool GoForward, double* dS, d
ouble* Range, double* dE) {

  if(debug_fit) {cout <<" InoTrackFitAlg : Swim "<< Plane<<" "<<NewPlane<<endl;}

  // Initialisations
  // customize for bfield scaling.
  //  BField * bf = new BField(*vldc,-1,0);
  SwimSwimmer* myswimmer = new SwimSwimmer(fabs(LayerThickness*(Plane-NewPlane))
, 0.5*LayerThickness); //*vldc,bf);
  if(debug_fit) { cout <<" InoTrackFitAlg : Swim "<< Plane<<" "<<NewPlane<<endl;}

  //  if(UseGeoSwimmer) GeoSwimmer::Instance()->Initialize(*vldc);

  //  double invSqrt2 = pow(1./2.,0.5);
  double charge = 0.;
  bool done = false;

  if(fabs(StateVector[4])>1.e-10) {

    double modp = fabs(1./StateVector[4]);

    // Fix, to account for fact the cosmic muons could move in direction of nega
tive z
    if(ZIncreasesWithTime==false) {modp=-modp;}

    double dsdz = pow((1.+pow(StateVector[2],2)+pow(StateVector[3],2)),0.5);
```



```
    double angle = 0;
    double ct=cos(angle);
    double st=sin(angle);

    //    double dxdz = invSqrt2*(StateVector[2]-StateVector[3]);
    //    double dydz = invSqrt2*(StateVector[2]+StateVector[3]);
    double dxdz = ct*StateVector[2]-st*StateVector[3];
    double dydz = st*StateVector[2]+ct*StateVector[3];

    // Set up current muon details
    if(StateVector[4]>0.) charge = 1.;
    else if(StateVector[4]<0.) charge = -1.;

    TVector3 position(ct*StateVector[0]-st*StateVector[1],
                      st*StateVector[0]+ct*StateVector[1],
                      ZPosLayer[Plane]); //SlcClustData[Plane][0].csh->GetZPos()
);

    TVector3 momentum(modp*(dxdz/dsdz),
                      modp*(dydz/dsdz),
                      modp/dsdz);

    //    TVector3 bfield = bf->GetBField(position);
    //TVector3 bfield(1.,1.,0.); //GMA-magnetic field  //AAR: commented out
    //    TVector3 bfield(1.5,0.,0.); //GMA-magnetic field
    //bave += TMath::Sqrt(bfield[0]*bfield[0]+bfield[1]*bfield[1]+bfield[2]*bfie
ld[2]); //AAR: commented out
    //    bave += pow(bfield[0]*bfield[0]+bfield[1]*bfield[1]+bfield[2]*bfield[2
],0.5);

    //nbfield++;   //AAR: commented out

    SwimParticle muon(position,momentum);
    muon.SetCharge(charge);
    //    cout <<"charge === "<<charge<<" "<<momentum.X()<<" "<<momentum.Y()<<"
"<<momentum.Z()<<" "<<position.X()<<" "<<position.Y()<<" "<<position.Z()<<" "<<d
xdz<<" "<<dydz<<" "<<dsdz<<" st "<<StateVector[0]<<" "<<StateVector[1]<<" "<<Sta
teVector[2]<<" "<<StateVector[3]<<" "<<StateVector[4]<<" "<<muon.GetMomentum().Z
()<<endl;

    //GMA    SwimZCondition zc(ZPosLayer[NewPlane]); //SlcClustData[NewPlane][0]
.csh->GetZPos());
    // Do the swim, accounting for direction of motion w.r.t time too
    if( (GoForward==true && ZIncreasesWithTime==true)  || (GoForward==false && Z
IncreasesWithTime==false) ) {
      if(UseGeoSwimmer) {
        //        done = GeoSwimmer::Instance()->SwimForward(muon,ZPosLayer[NewP
lane]); //SlcClustData[NewPlane][0].csh->GetZPos());
      } else {
        done = myswimmer->SwimForward(muon,t_bave);
      }
    } else if( (GoForward==true && ZIncreasesWithTime==false)  || (GoForward==fa
lse && ZIncreasesWithTime==true) ) {
      if(UseGeoSwimmer) {
        //        done = GeoSwimmer::Instance()->SwimBackward(muon,ZPosLayer[New
Plane]); //SlcClustData[NewPlane][0].csh->GetZPos());
      } else {
        done = myswimmer->SwimBackward(muon,t_bave);
      }
    }


    bave += t_bave;  //AAR:  added
    nbfield++;       //AAR:  added
    if(done==true) {

      if(muon.GetDirection().Z()!=0. && muon.GetMomentumModulus()!=0.) {
        angle = 0;
        ct=cos(angle);
```

```
        st=sin(angle);
        Output[0]=(st*muon.GetPosition().Y()+ct*muon.GetPosition().X());
        Output[1]=(ct*muon.GetPosition().Y()-st*muon.GetPosition().X());
        Output[2]=(st*(muon.GetDirection().Y()/muon.GetDirection().Z())+ct*(muon
.GetDirection().X()/muon.GetDirection().Z()));
        Output[3]=(ct*(muon.GetDirection().Y()/muon.GetDirection().Z())-st*(muon
.GetDirection().X()/muon.GetDirection().Z()));
        Output[4]=muon.GetCharge()/muon.GetMomentumModulus();

        Output[5]= StateVector[5];
        // Get range and dS from the Swimmer
        if(dS) {*dS=muon.GetS();}
        if(Range) {*Range=muon.GetRange();}
        if(dE){*dE=muon.GetMomentumModulus()-momentum.Mag();}

        //GMA put this more elegantly
        fTrackCand->fdS[NewPlane] =muon.GetS();
        fTrackCand->fRange[NewPlane] =muon.GetRange();

      } else {done=false;}
    }

  } else {
    // If infinite momentum, use straight line extrapolation
    double delz = LayerThickness;
    //    cout <<"delz "<< delz<<endl;
    if (SlcClustData[NewPlane].size()>0 && SlcClustData[Plane].size()>0) {
      //      delz = (SlcClustData[NewPlane][0].csh->GetZPos()-SlcClustData[Plan
e][0].csh->GetZPos());
      delz = ZPosLayer[NewPlane] - ZPosLayer[Plane];
    }

    //    cout <<"delz "<< delz<<endl;

    Output[0]=StateVector[0] + StateVector[2]*delz;
    Output[1]=StateVector[1] + StateVector[3]*delz;
    Output[2]=StateVector[2];
    Output[3]=StateVector[3];
    Output[4]=StateVector[4];
    Output[5]=StateVector[5];

    done=true;
  }
  //cout <<" Input S1 "<< StateVector[0]<<" "<<StateVector[1]<<" "<<StateVector[
2]<<" "<<StateVector[3]<<" "<<StateVector[4]<<endl;
  //cout <<" Output S1 "<< Output[0]<<" "<<Output[1]<<" "<<Output[2]<<" "<<Outpu
t[3]<<" "<<Output[4]<<endl;
  delete myswimmer;
  //  delete bf;
  return done;
}


bool InoTrackFitAlg::Swim(double* StateVector, double* Output, const int Plane,
                          const double zend, const bool GoForward, double* dS, d
ouble* Range, double* dE) {

  SwimSwimmer* myswimmer = new SwimSwimmer(zend, 0.5*LayerThickness); //*vldc,bf
);

  double charge = 0.;
  bool done = false;

  if(fabs(StateVector[4])>1.e-10) {
    double modp = fabs(1./StateVector[4]);

    if(ZIncreasesWithTime==false) {modp=-modp;}

    double dsdz = pow((1.+pow(StateVector[2],2)+pow(StateVector[3],2)),0.5);
```



```

    double angle = 0;
    double ct=cos(angle);
    double st=sin(angle);

    double dxdz = ct*StateVector[2]-st*StateVector[3];
    double dydz = st*StateVector[2]+ct*StateVector[3];

    // Set up current muon details
    if(StateVector[4]>0.) charge = 1.;
    else if(StateVector[4]<0.) charge = -1.;

    TVector3 position(ct*StateVector[0]-st*StateVector[1],
                      st*StateVector[0]+ct*StateVector[1],
                      ZPosLayer[Plane]); //SlcClustData[Plane][0].csh->GetZPos()
);

    TVector3 momentum(modp*(dxdz/dsdz),
                      modp*(dydz/dsdz),
                      modp/dsdz);

    //TVector3 bfield(1.,1.,0.); //GMA-magnetic field  //AAR  commented out
    //    TVector3 bfield(1.5,0.,0.); //GMA-magnetic field
    //bave += //TMath::Sqrt(bfield[0]*bfield[0]+bfield[1]*bfield[1]+bfield[2]*bf
ield[2]); //AAR commented out
    //    bave += pow(bfield[0]*bfield[0]+bfield[1]*bfield[1]+bfield[2]*bfield[2
],0.5);
    //nbfield++; //AAR commented out

    SwimParticle muon(position,momentum);
    muon.SetCharge(charge);

    //GMA    SwimZCondition zc(ZPosLayer[NewPlane]); //SlcClustData[NewPlane][0]
.csh->GetZPos());
    // Do the swim, accounting for direction of motion w.r.t time too
    if( (GoForward==true && ZIncreasesWithTime==true)  || (GoForward==false && Z
IncreasesWithTime==false) ) {
      done = myswimmer->SwimForward(muon,t_bave);
    }
    else if( (GoForward==true && ZIncreasesWithTime==false)  || (GoForward==fals
e && ZIncreasesWithTime==true) ) {
      done = myswimmer->SwimBackward(muon,t_bave);
    }
    bave += t_bave;  //AAR:  added
    nbfield++;       //AAR:  added
    if(done==true) {
      if(muon.GetDirection().Z()!=0. && muon.GetMomentumModulus()!=0.) {
        angle = 0;
        ct=cos(angle);
        st=sin(angle);
        Output[0]=(st*muon.GetPosition().Y()+ct*muon.GetPosition().X());
        Output[1]=(ct*muon.GetPosition().Y()-st*muon.GetPosition().X());
        Output[2]=(st*(muon.GetDirection().Y()/muon.GetDirection().Z())+ct*(muon
.GetDirection().X()/muon.GetDirection().Z()));
        Output[3]=(ct*(muon.GetDirection().Y()/muon.GetDirection().Z())-st*(muon
.GetDirection().X()/muon.GetDirection().Z()));
        Output[4]=muon.GetCharge()/muon.GetMomentumModulus();
        Output[5]= StateVector[5];

        // Get range and dS from the Swimmer
        if(dS) {*dS=muon.GetS();}
        if(Range) {*Range=muon.GetRange();}
        if(dE){*dE=muon.GetMomentumModulus()-momentum.Mag();}

        //GMA put this more elegantly
        fTrackCand->SetdSExtra(muon.GetS());
        fTrackCand->SetRangeExtra(muon.GetRange());

      } else {done=false;}
```

```
    }
  } else {
    // If infinite momentum, use straight line extrapolation
    Output[0]=StateVector[0] + StateVector[2]*zend;
    Output[1]=StateVector[1] + StateVector[3]*zend;
    Output[2]=StateVector[2];
    Output[3]=StateVector[3];
    Output[4]=StateVector[4];
    Output[5]=StateVector[5];

    done=true;
  }
  //cout <<" Output "<< Output[0]<<" "<<Output[1]<<" "<<Output[2]<<" "<<Output[3
]<<" "<<Output[4]<<endl;
  delete myswimmer;
  //  delete bf;
  return done;
}

bool InoTrackFitAlg::Swim(double* StateVector, double* Output, const double zzz,

                          const int NewPlane,const bool GoForward, double* dS, d
ouble* Range, double* dE) {
  if(debug_fit) { cout <<" InoTrackFitAlg : Swim, specified starting Z" << endl;}

  // Initialisations
  // customize for bfield scaling.
  //  BField * bf = new BField(*vldc,-1,0);

  // GMA Need to extrace proper Z values for a plane
  SwimSwimmer* myswimmer = new SwimSwimmer(fabs(LayerThickness*NewPlane-zzz), 0.
5*LayerThickness);

  //  if(UseGeoSwimmer) GeoSwimmer::Instance()->Initialize(*vldc);

  //  double invSqrt2 = pow(1./2.,0.5);
  double charge = 0.;
  bool done = false;

  if(fabs(StateVector[4])>1.e-10) {
    double modp = fabs(1./StateVector[4]);

    // Fix, to account for fact the cosmic muons could move in direction of nega
tive z
    if(ZIncreasesWithTime==false) {modp=-modp;}

    double dsdz = pow((1.+pow(StateVector[2],2)+pow(StateVector[3],2)),0.5);
    double angle=0;
    double ct=cos(angle);
    double st=sin(angle);
    double dxdz = ct*StateVector[2]-st*StateVector[3];
    double dydz = st*StateVector[2]+ct*StateVector[3];

    // Set up current muon details
    if(StateVector[4]>0.) charge = 1.;
    else if(StateVector[4]<0.) charge = -1.;

    TVector3 position(ct*StateVector[0]-st*StateVector[1],
                      st*StateVector[0]+ct*StateVector[1],
                      zzz);

    TVector3 momentum(modp*(dxdz/dsdz),
                      modp*(dydz/dsdz),
                      modp/dsdz);
    SwimParticle muon(position,momentum);
    muon.SetCharge(charge);
    //GMA    SwimZCondition zc(ZPosLayer[NewPlane]); //SlcClustData[NewPlane][0]
.csh->GetZPos());
```



```
    // Do the swim, accounting for direction of motion w.r.t time too
    if( (GoForward==true && ZIncreasesWithTime==true)  || (GoForward==false && Z
IncreasesWithTime==false) ) {
      if(UseGeoSwimmer) {
        //        done = GeoSwimmer::Instance()->SwimForward(muon,ZPosLayer[NewP
lane]); //SlcClustData[NewPlane][0].csh->GetZPos());
      } else {
        done = myswimmer->SwimForward(muon,t_bave);
      }
    }
    else if( (GoForward==true && ZIncreasesWithTime==false)  || (GoForward==fals
e && ZIncreasesWithTime==true) ) {
      if(UseGeoSwimmer) {
        //        done = GeoSwimmer::Instance()->SwimBackward(muon,ZPosLayer[New
Plane]); //SlcClustData[NewPlane][0].csh->GetZPos());

      } else {
        done = myswimmer->SwimBackward(muon,t_bave);
      }
    }
    if(done==true) {
      angle=0;
      ct=cos(angle);
      st=sin(angle);

      if(muon.GetDirection().Z()!=0. && muon.GetMomentumModulus()!=0.) {
        Output[0]=(st*muon.GetPosition().Y()+ct*muon.GetPosition().X());
        Output[1]=(ct*muon.GetPosition().Y()-st*muon.GetPosition().X());
        Output[2]=(st*(muon.GetDirection().Y()/muon.GetDirection().Z())+ct*(muon
.GetDirection().X()/muon.GetDirection().Z()));
        Output[3]=(ct*(muon.GetDirection().Y()/muon.GetDirection().Z())-st*(muon
.GetDirection().X()/muon.GetDirection().Z()));
        Output[4]=muon.GetCharge()/muon.GetMomentumModulus();
        Output[5]=StateVector[5];
        // Get range and dS from the Swimmer
        if(dS) {*dS=muon.GetS();} if(Range) {*Range=muon.GetRange();} if(dE){*dE
=muon.GetMomentumModulus()-momentum.Mag();}
      } else {done=false;}
    }
  } else {
    // If infinite momentum, use straight line extrapolation
    double delz = (ZPosLayer[NewPlane] -zzz); //SlcClustData[NewPlane][0].csh->G
etZPos()-z);
    Output[0]=StateVector[0] + StateVector[2]*delz;
    Output[1]=StateVector[1] + StateVector[3]*delz;
    Output[2]=StateVector[2];
    Output[3]=StateVector[3];
    Output[4]=StateVector[4];
    Output[5]=StateVector[5];
    done=true;
  }
  //cout <<" Output "<< Output[0]<<" "<<Output[1]<<" "<<Output[2]<<" "<<Output[3
]<<" "<<Output[4]<<endl;
  delete myswimmer;
  //  delete bf;

  return done;
}
void InoTrackFitAlg::ResetCovarianceMatrix() {
  // Simple method reset variables/arrays to allow propagation again

  DeltaPlane=0; DeltaZ=0;
  GetInitialCovarianceMatrix(false);
}

void InoTrackFitAlg::GetInitialCovarianceMatrix(const bool FirstIteration1) {
  if (debug_fit) { cout <<" InoTrackFitAlg : GetInitialCovarianceMatrix " << FirstIteration1
<< endl;}
```

```
  if(FirstIteration1==true) {
    for(int ij=0; ij<5; ++ij) {
      for(int jk=0; jk<5; ++jk) {
        C_k_minus[ij][jk]=0.0;

        if (jk == ij) {
          C_k_minus[ij][jk]=1.e5;
        }
      }
    }
  } else if (FirstIteration1==false) {
    for (int ij = 0; ij < 5; ++ij) {
      for (int jk = 0; jk < 5; ++jk) {
        C_k_minus[ij][jk] = 1.e2*C_k_minus[ij][jk];
      }
    }
  }
  // Display
  if(debug_fit) {
    cout<<"---------------------------------------------------------------- "<<endl;
    cout << "Initial covariance matrix" << endl;
    for(int ij=0; ij<5; ++ij) {
      for(int jk=0; jk<5; ++jk) {
        cout << C_k_minus[ij][jk] << " ";
      }
      cout << endl;
    }
    cout<<"---------------------------------------------------------------- "<<endl;
  }
}

void InoTrackFitAlg::CalcKalmanGain(double *x__minus,const int NewPlane) {
  if(debug_fit) {
    cout <<" InoTrackFitAlg : CalcKalmanGain" <<x__minus[0]<<" "<<x__minus[1]<<" "<<x__mi
nus[2]<<" "<<x__minus[3]<<" "<<x__minus[4]<< endl;
  }
  // K_k = C_k_intermediate * H_k^T * ( V_k + H_k * C_k_intermediate * H_k^T )^-
1
  //                                                                    ( = A1_k
)^-1
  //                                                                    = B2_k;

  double hitX = 0.0;
  double hitY = 0.0;
  double A1_k[2][2];
  double B1_k[2][2];
  bool OutLier=true;

  if (NewPlane==OtLStrip) {
    //cout<<"strip identified: "<<NewPlane<<endl;
    OutLier = true;
  }

  for (int ij=0; ij<2; ij++) {
    for (int jk=0; jk<2; jk++) {
      A1_k[ij][jk] = 0;
      B1_k[ij][jk] = 0;
    }
  }
  // H_k has only one non-zero element, so we can reduce matrix multiplication r
equired

  int PlaneView = TrkClustsData[NewPlane][0].PlaneView;

  if(PlaneView%2==0) {
    hitX                = TrkClustsData[NewPlane][0].XPos;
    A1_k[0][0]  = C_k_intermediate[0][0];
    if (fMT==false) {
```



```
      A1_k[0][0] += 1.00*TrkClustsData[NewPlane][0].XPosErrSq;  // Add uncertain
ty in measurement::xx 3.20133e-05
    } else {
      A1_k[0][0] += 1.00*TrkClustsData[NewPlane][0].XPosErrSq;
    }
  }
  if (PlaneView >0) {
    hitY                = TrkClustsData[NewPlane][0].YPos;
    A1_k[1][1]  = C_k_intermediate[1][1];
    if (fMT==false) {
      A1_k[1][1] += 1.00*TrkClustsData[NewPlane][0].YPosErrSq;  // Add uncertain
ty in measurement::yy 3.20133e-05
    } else {
      A1_k[1][1] += 1.00*TrkClustsData[NewPlane][0].YPosErrSq;
    }
  }

  if (PlaneView==2) {
    A1_k[0][1] = C_k_intermediate[0][1];
    A1_k[1][0] = C_k_intermediate[1][0];
  }

  if (OutLier == false) {
    //cout<<"exercised"<<endl;
    A1_k[0][0]  = C_k_intermediate[0][0] - TrkClustsData[NewPlane][0].XPosErrSq;
    A1_k[1][1]  = C_k_intermediate[1][1] - TrkClustsData[NewPlane][0].YPosErrSq;
  }

  double determinant = A1_k[0][0]*A1_k[1][1] - A1_k[0][1]*A1_k[1][0];
  // cout<<"det = "<<determinant<<endl;
  // Inverse matrix;
  // cout <<" InoTrackFitAlg : V_k " << NewPlane<<" "<<TrkClustsData[NewPlane][0
].XPosErrSq <<" " <<TrkClustsData[NewPlane][0].YPosErrSq<<" "<<PlaneView<<" "<<d
eterminant<<" "<<A1_k[0][0]<<" "<<A1_k[0][1]<<" "<<A1_k[1][0]<<" "<<A1_k[1][1]<<
endl;
  if (determinant !=0) {
    B1_k[0][0] =  A1_k[1][1]/determinant;
    B1_k[0][1] = -A1_k[0][1]/determinant;
    B1_k[1][0] = -A1_k[1][0]/determinant;
    B1_k[1][1] =  A1_k[0][0]/determinant;

    for (int ij=0; ij<5; ++ij) {
      for (int jk =0; jk<2; jk++) {
        K_k[ij][jk]=0;
        for (int kl=0; kl<5; ++kl) {
          for (int lm=0; lm<2; lm++) {
            K_k[ij][jk] +=C_k_intermediate[ij][kl]*H_k[lm][kl]*B1_k[lm][jk];
          }
        }
      }
    }

    for (int ij = 0; ij < 5; ij++) {
      for (int jk = 0; jk < 2; jk++) {
        //cout<<"K_k["<<ij<<"]["<<jk<<"] "<<K_k[ij][jk]<<endl;
      }
    }

  } else {
    //   cout <<" InoTrackFitAlg : V_k + (H_k * C_k_intermediate * H_k_transpose
) is zero!" << endl;
  }
}

void InoTrackFitAlg::UpdateCovMatrix(const int NewPlane) {
  // C_k = (Identity - (K_k * H_k) ) * C_k_intermediate
  // if(debug_fit) {cout <<"*****InoTrackFitAlg : UpdateCovMatrix*****" << endl;
}
```

```
  double KH[5][5] = {{0.0}};
  double Ck[5][5] = {{0.0}};
  double Vk[2][2] = {{0.0}};
  double KVKt[5][5]={{0.0}};

  // Evaluate KH = K_k * H_k
  for (int ij = 0; ij  <5; ij++) {
    for (int jk = 0; jk < 5; jk++) {
      KH[ij][jk] = 0;
      for (int kl = 0; kl < 2; kl++) {
        KH[ij][jk] += K_k[ij][kl]*H_k[kl][jk];
      }
    }
  }

  // Now, find (I - KH) = (I - K_k*H_k): Redefine KH:=(I - KH)
  for (int ij = 0; ij < 5; ij++) {
    for (int jk = 0; jk < 5; jk++) {
      KH[ij][jk] = Identity[ij][jk] - KH[ij][jk];
    }
  }

  // The expression: C_k = (I-KH) * C_p has known issues with rounding errors
  // They may lead to -ve elements in the diagonal of error covariance
  // Use the following expression: C_k = (I-KH) * C_p * (I-KH)^ + K * V * K^
  // This expression is the sum of two positive definite matrices

  // First find K * V * K^
  //-------------------------------------------------------------------
  // definition of Vk:
  if (fMT==false) {
    Vk[0][0] = 1.0*TrkClustsData[NewPlane][0].XPosErrSq;        //cout<<"Vk[0][0
]"<<Vk[0][0]<<endl;
    Vk[1][1] = 1.0*TrkClustsData[NewPlane][0].YPosErrSq;        //cout<<"Vk[1][1
]"<<Vk[1][1]<<endl;
  } else {
    Vk[0][0] =1.0*TrkClustsData[NewPlane][0].XPosErrSq; //cout<<"Vk[0][0]"<<Vk[0
][0]<<endl;
    Vk[1][1] =1.0*TrkClustsData[NewPlane][0].YPosErrSq; //cout<<"Vk[1][1]"<<Vk[1
][1]<<endl;
  }

  //cout<<"..................................................."<<endl;

  // Obtaining K * V * K^
  for (int ij = 0; ij < 5; ij++) {
    for (int jk = 0; jk < 5; jk++) {
      KVKt[ij][jk] = 0;
      for (int kl = 0; kl < 2; kl++) {
        for (int lm = 0; lm < 2; lm++) {
          KVKt[ij][jk] += K_k[ij][lm] * Vk[lm][kl] * K_k[jk][kl];
        }
      }
    }
  }
  //-------------------------------------------------------------------
  // Then, find (I-KH) * C_p * (I-KH)^
  //-------------------------------------------------------------------
  for (int ij = 0; ij < 5; ij++) {
    for (int jk = 0; jk < 5; jk++) {
      Ck[ij][jk] = 0;
      for (int kl = 0; kl < 5; kl++) {
        for (int lm = 0; lm < 5; lm++) {
          Ck[ij][jk] += KH[ij][lm] * C_k_intermediate[lm][kl] * KH[jk][kl];
        }
      }
    }
  }
  //-------------------------------------------------------------------
```



```
  // Hence, find C_k (filtered covariance)
  //-------------------------------------------------------------------
  for(int ij = 0; ij < 5; ij++) {
    for(int jk = 0; jk < 5; jk++) {
      C_k[ij][jk] = Ck[ij][jk] + KVKt[ij][jk];
    }
  }

  if(debug_fit) {cout <<" InoTrackFitAlg : UpdateCovMatrix ends" << endl;}
}
void InoTrackFitAlg::MoveArrays(const int NewPlane, const bool GoForward) {
  // Move k to k-1 ready to consider next clust
  for (int ij=0; ij<5; ++ij) {
    for (int jk=0; jk<5; ++jk) {
      C_k_minus[ij][jk]=0;
      C_k_minus[ij][jk]=C_k[ij][jk];
    }
  }

  for (int ij=0; ij<5; ++ij) {
    x_k_minus[ij]= 0.0;
    x_k_minus[ij]=x_k[ij];
  }

  //  double chisquare=0.; GMA14
  double m_k[2]={0.0};
  double sigma_xx=0.0;
  double sigma_yy=0.0;

  m_k[0]        = TrkClustsData[NewPlane][0].XPos;
  m_k[1]        = TrkClustsData[NewPlane][0].YPos;

  if (fMT==false) {
    sigma_xx= TrkClustsData[NewPlane][0].XPosErrSq;
    sigma_yy= TrkClustsData[NewPlane][0].YPosErrSq;
  } else {
    sigma_xx=1.0*TrkClustsData[NewPlane][0].XPosErrSq;
    sigma_yy=1.0*TrkClustsData[NewPlane][0].YPosErrSq;
  }

  double r_k[2] = {0.0};
  r_k[0]                        = m_k[0]-x_k[0];
  r_k[1]                        = m_k[1]-x_k[1];

  double R_kDet = (sigma_xx-C_k[0][0])*(sigma_yy-C_k[1][1])-C_k[0][1]*C_k[1][0];
  double R_k[2][2]= {{0.0}};
  R_k[0][0]             = (1.0/R_kDet)*(sigma_yy-C_k[1][1]);
  R_k[0][1]             = (1.0/R_kDet)*C_k[0][1];
  R_k[1][0]             = (1.0/R_kDet)*C_k[1][0];
  R_k[1][1]             = (1.0/R_kDet)*(sigma_xx-C_k[0][0]);

  double chi2   = R_k[0][0]*r_k[0]*r_k[0] + R_k[0][1]*r_k[0]*r_k[1] + R_k[1][0]*
r_k[1]*r_k[0] + R_k[1][1]*r_k[1]*r_k[1];

  if ((FirstIteration) && (GoForward==false) && (chi2 > 5.0)) {
    OtLStrip=NewPlane;
    //cout<<NewPlane<<" fed into OltStrip"<<endl;
  }

  if ((LastIteration) && (GoForward == true)) {
    ChiSquare += chi2;
  }
  if(debug_fit) {
    cout <<" InoTrackFitAlg : MoveArrays end" << endl;
  }
}

void InoTrackFitAlg::StoreFilteredData(const int NewPlane) {
  // Store the data required for matching Kalman output data to strips
```

```
  if(debug_fit) { cout <<" InoTrackFitAlg : StoreFilteredData" << endl;}
  for (unsigned ij=0; ij<FilteredData[NewPlane].size(); ij++) {
    if (FilteredData[NewPlane][ij].x_k5==0) {
      FilteredData[NewPlane].erase(FilteredData[NewPlane].begin()+ij);
      ij-- ;
    }
  }

  FiltDataStruct temp;

  temp.x_k0=x_k[0]; temp.x_k1=x_k[1];
  temp.x_k2=x_k[2]; temp.x_k3=x_k[3];
  temp.x_k4=x_k[4];

  temp.x_k5=int(x_k[5]);
  temp.x_k6=true;
  //  FilteredData[NewPlane].clear();
  FilteredData[NewPlane].push_back(temp);
}

void InoTrackFitAlg::StoreFilteredData_sr(const int NewPlane, double* prediction
, bool str) {
  // Store the data required for matching Kalman output data to strips
  if(debug_fit) {  cout <<"InoTrackFitAlg : StoreFilteredData_sr"<<endl;}
  for (unsigned ij=0; ij<FilteredData[NewPlane].size(); ij++) {
    if (FilteredData[NewPlane][ij].x_k5==0) {
      FilteredData[NewPlane].erase(FilteredData[NewPlane].begin()+ij);
      ij-- ;
    }
  }

  FiltDataStruct temp;
  temp.x_k0=prediction[0];
  temp.x_k1=prediction[1];
  temp.x_k2=prediction[2];
  temp.x_k3=prediction[3];
  temp.x_k4=prediction[4];
  temp.x_k5=0;
  temp.x_k6=str;
  // FilteredData[NewPlane].clear();
  FilteredData[NewPlane].push_back(temp);
  if(debug_fit) {cout <<"InoTrackFitAlg : StoreFilteredData_sr end"<<endl;}
}

void InoTrackFitAlg::SetTrackProperties(double* x_xk) {
  // Carry out the assignment of variables to the new fitted track
  //  if(debug_fit) {cout <<"InoTrackFitAlg: SetTrackProperties"<<endl;}
  // -------------------------------------------------------------------
  if(x_xk[4]>0.) {
    fTrackCand->SetEMCharge(+1.0);
  } else if(x_xk[4]<0.) {
    fTrackCand->SetEMCharge(-1.0);
  }
  // -------------------------------------------------------------------
  // Vtx & End planes of the fit
  int VtxPlane;
  int EndPlane;

  if(ZIncreasesWithTime==true) {
    VtxPlane=MinPlane;
    EndPlane=MaxPlane;
  } else {
    VtxPlane=MaxPlane;
    EndPlane=MinPlane;
  }

  // cout<<"ZIncreasesWithTime = "<<ZIncreasesWithTime<<endl;
  // cout<<"VtxPlane = "<<VtxPlane<<", EndPlane = "<<EndPlane<<endl;
  // cout<<"1./x_xk[4] = "<<1./x_xk[4]<<endl;
```



```
  fTrackCand->SetVtxZ(ZPosLayer[VtxPlane]); //SlcClustData[VtxPlane][0].csh->Get
ZPos());
  fTrackCand->SetVtxPlane(VtxPlane);
  fTrackCand->SetVtxRPCmod(VtxPlane);
  fTrackCand->SetEndZ(ZPosLayer[EndPlane]); //SlcClustData[EndPlane][0].csh->Get
ZPos());
  fTrackCand->SetEndPlane(EndPlane);
  fTrackCand->SetEndRPCmod(EndPlane);
  // -------------------------------------------------------------------
  // Input Parameters to KF
  fTrackCand->SetVtxXX(xxin);
  fTrackCand->SetVtxYY(yyin);
  fTrackCand->SetVtxTX(txin);
  fTrackCand->SetVtxTY(tyin);
  // -------------------------------------------------------------------
  // Fit parameters at vtx
  //++++++++++++++++++++++++++++Vtx+++++++++++++++++++++++++++++++

  fTrackCand->SetVtxU(x_xk[0]);
  fTrackCand->SetVtxV(x_xk[1]);
  fTrackCand->SetVtxdU(x_xk[2]);
  fTrackCand->SetVtxdV(x_xk[3]);

  double dsdz;
  dsdz=sqrt(1.0 + x_xk[2]*x_xk[2] + x_xk[3]*x_xk[3]);
  if(ZIncreasesWithTime==false) {dsdz = - dsdz;}

  fTrackCand->SetTheta(acos(1./dsdz));

  fTrackCand->SetPhi(atan2(x_xk[3],x_xk[2]));

  fTrackCand->SetFCPC(FCorPC);

  // cout<<"FCorPC = "<<FCorPC<<", fTrackCand->SetFCPC(FCorPC); = "<<fTrackCand-
>GetFCPC()<<endl;
  if (x_xk[4]!=0) {
    fTrackCand->SetMomentum(1./x_xk[4]);
  }

  double therr = 0.0;
  double pherr = 0.0;

  // Following formulae for therr & pherr developed from txerr and tyerr by stan
dard error propagation
  therr = (1./(1. + x_xk[2]*x_xk[2] + x_xk[3]*x_xk[3]))*(sqrt(x_xk[2]*x_xk[2]*Vt
xCov[2]+x_xk[3]*x_xk[3]*VtxCov[3]+2*x_xk[2]*x_xk[3]*C_k[2][3])/sqrt(x_xk[2]*x_xk
[2]+x_xk[3]*x_xk[3]));
  pherr = sqrt(x_xk[2]*x_xk[2]*VtxCov[3] + x_xk[3]*x_xk[3]*VtxCov[2] - 2*x_xk[2]
*x_xk[3]*C_k[2][3])/(x_xk[2]*x_xk[2] + x_xk[3]*x_xk[3]);

  fTrackCand->SetThErr(therr);
  fTrackCand->SetPhErr(pherr);

  // Errors on vtx positions, angles and q/p
  fTrackCand->SetVtxUError(pow(VtxCov[0],0.5));
  fTrackCand->SetVtxVError(pow(VtxCov[1],0.5));
  fTrackCand->SetVtxdUError(pow(VtxCov[2],0.5));
  fTrackCand->SetVtxdVError(pow(VtxCov[3],0.5));
  fTrackCand->SetVtxQPError(pow(VtxCov[4],0.5));
  // -------------------------------------------------------------------
  //++++++++++++++++++++++++++++End+++++++++++++++++++++++++++++++

  // Vtx and end direction cosines
  dsdz=pow(1.+pow(FilteredData[VtxPlane][0].x_k2,2)+pow(FilteredData[VtxPlane][0
].x_k3,2),0.5);

  if(ZIncreasesWithTime==false) {dsdz=-dsdz;}
  fTrackCand->SetVtxDirCosU(FilteredData[VtxPlane][0].x_k2/dsdz);
```

```
  fTrackCand->SetVtxDirCosV(FilteredData[VtxPlane][0].x_k3/dsdz);
  fTrackCand->SetVtxDirCosZ(1./dsdz);
  if (FilteredData[VtxPlane][0].x_k4!=0) {
    fTrackCand->SetMomentumCurve(1./FilteredData[VtxPlane][0].x_k4);
  } else {
    fTrackCand->SetMomentumCurve(-1000.);
  }

  fTrackCand->SetDirCosU(FilteredData[VtxPlane][0].x_k2/dsdz);
  fTrackCand->SetDirCosV(FilteredData[VtxPlane][0].x_k3/dsdz);
  fTrackCand->SetDirCosZ(1./dsdz);
  // Fit information in the end plane
  fTrackCand->SetEndU(EndState[0]);
  fTrackCand->SetEndV(EndState[1]);
  fTrackCand->SetEndDirCosU(EndState[2]);
  fTrackCand->SetEndDirCosV(EndState[3]);

  dsdz=sqrt(1.0 + EndState[2]*EndState[2] + EndState[3]*EndState[3]);
  if(ZIncreasesWithTime==false) {dsdz=-dsdz;}
  fTrackCand->SetEndDirCosZ(1./dsdz);
  fTrackCand->SetEndMomentumCurve(1./EndState[4]);
  fTrackCand->SetEndQP(EndState[4]);

  // Errors on end positions, angles and q/p
  fTrackCand->SetEndUError(pow(EndCov[0],0.5));
  fTrackCand->SetEndVError(pow(EndCov[1],0.5));
  fTrackCand->SetEnddUError(pow(EndCov[2],0.5));
  fTrackCand->SetEnddVError(pow(EndCov[3],0.5));
  fTrackCand->SetEndQPError(pow(EndCov[4],0.5));
  // ------------------------------------------------------------------
  // ------------------------------------------------------------------
  fTrackCand->SetChi2(ChiSquare/(2*nHit-5));
  fTrackCand->SetMomentumdS(GPL);
  fTrackCand->SetMomentumRange(RNG);
  // ------------------------------------------------------------------
  // ------------------------------------------------------------------

  // More variables to be set, in order to ensure compatibility
  fTrackCand->SetNTrackStrip(fFinderTrack->ClustsInTrack.size());
  fTrackCand->SetNIterate(NIter);
  fTrackCand->SetNSwimFail(TotalNSwimFail);

  // Obtain "fitting data" for the final track strips
  // for (unsigned ij=0; ij<nLayer; ++ij) {TrkClustsData[ij].clear();}
  // GetFitData(MinPlane,MaxPlane);

  // Set tpos error and Calculate chi2, NDOF
  double Chi2=0; double Chi2Contrib=0; int NDOF=0; double FilteredXPos=0; double
 FilteredYPos=0;
  double momdS=0; double momRange=0;

  double sxy=0; // y (distance) = c t + shift
  double sx=0;  // y = distance
  double sy=0;  // x = time
  double sn=0;
  double sx2=0;
  for (unsigned ijk=0; ijk<fTrackCand->GetClusterEntries(); ijk++) {
    int ij = fTrackCand->ClustsInTrack[ijk]->GetZPlane();
    // for(int ij=MinPlane; ij<=MaxPlane; ++ij) {
    if (ij <=int(nLayer))  {
      if(TrkClustsData[ij].size()>0) {
        if(TrkClustsData[ij][0].XPosErrSq>0. || TrkClustsData[ij][0].YPosErrSq>0
.) {
          fTrackCand->SetTrackPointXError(ij,pow(TrkClustsData[ij][0].XPosErrSq,
0.5));
          fTrackCand->SetTrackPointYError(ij,pow(TrkClustsData[ij][0].YPosErrSq,
0.5));
          momdS += max(0., double(fTrackCand->GetdS(ij)));
          momRange += max(0.,double(fTrackCand->GetRange(ij)));
```



```
          // if ((!ZIncreasesWithTime && ijk>0) || (ZIncreasesWithTime && ijk <f
TrackCand->GetEntries()-1)) {
          sn +=1;
          sx += TrkClustsData[ij][0].cltime; //Look again for return track
          sy += momdS;
          sxy = momdS*TrkClustsData[ij][0].cltime;
          sx2 = (TrkClustsData[ij][0].cltime)*(TrkClustsData[ij][0].cltime);

          //      }

          Chi2Contrib = 0;

          for (unsigned jk=0; jk<TrkClustsData[ij].size() ;jk++) {
            if(TrkClustsData[ij][jk].PlaneView%2==0) {
              FilteredXPos=FilteredData[ij][jk].x_k0;
              Chi2Contrib += pow((TrkClustsData[ij][jk].XPos-FilteredXPos),2)/Tr
kClustsData[ij][jk].XPosErrSq;
              NDOF++;
            }
            if (TrkClustsData[ij][jk].PlaneView >=1) {
              FilteredYPos=FilteredData[ij][jk].x_k1;
              Chi2Contrib += pow((TrkClustsData[ij][jk].YPos-FilteredYPos),2)/Tr
kClustsData[ij][jk].YPosErrSq;
              NDOF++;
            }
          }
          fTrackCand->SetPlaneChi2(ij,Chi2Contrib);
          Chi2+=Chi2Contrib;
          //     NDOF++;
        }
      }
    } else {
      if (TrkClustsData[ij+shiftLa].size()>0) {
        momdS += fTrackCand->Get2dS(ij+shiftLa);
        momRange += fTrackCand->Get2Range(ij+shiftLa);
        sn +=1;
        sx += TrkClustsData[ij+shiftLa][0].cltime; //Look again for return track
        sy += momdS;
        sxy += momdS*TrkClustsData[ij+shiftLa][0].cltime;
        sx2 += (TrkClustsData[ij][0+shiftLa].cltime)*(TrkClustsData[ij][0].cltim
e);
        if(pAnalysis->isXtermOut==1) {
          //cout <<"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX "<<endl;
          //cout <<"ijk "<<ijk<<" "<<ij<<" "<<momdS<<" "<<momRange<<" "<<endl;
          //cout <<"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX "<<endl;
        }
        //isXterm
      }
    }
  }

  double velocity = 0;
  if (sn >0 && (sx2*sn - sx*sx) !=0) {
    velocity = 10*(sxy*sn - sx*sy)/(sx2*sn - sx*sx); // x 10^8m/s
  }

  if(pAnalysis->isXtermOut==1) {
    cout<<endl;
    //cout <<"chisq : Chi2 NDOF momdS momRange  velocity"<<endl;
    //cout <<"chisq "<<Chi2 <<" "<<NDOF<<" "<<momdS<<" "<<momRange<<" "<<velocit
y<< endl;
    //cout<<"-------------------------------------------------------------------
--------"<<endl;
  }
  //isXterm

  fTrackCand->Setcval(velocity);
  fTrackCand->SetNDOF(NDOF-5); // Number of constraints set to 5
```

```
  // fTrackCand->SetMomentum(momRange);
  // Assign U, V and q/p values
  for(int ij=MinPlane; ij<=MaxPlane;++ij) {
    //asm_170311
    if(FilteredData[ij].size()>0) {
      fTrackCand->SetU(ij,FilteredData[ij][0].x_k0);
      fTrackCand->SetV(ij,FilteredData[ij][0].x_k1);
      fTrackCand->SetPlaneQP(ij,FilteredData[ij][0].x_k4);
    }
  }

  // cout<<"------------------------------------------------------------------
-----"<<endl;
  // cout<<"Reconstructed P = "<<fTrackCand->GetMomentum()<<"  |  "<<"theta "<<f
TrackCand->GetTheta()<<endl;
  // //"  |  L "<<FinderPathLength<<"  |  MaxPlane "<<MaxPlane<<"  |  MinPlane "
<<MinPlane<<"  |  #hits "<<n<<"  |  chisq/dof "<<ChiSquare/(2*n-5)<<endl;
  // cout<<"------------------------------------------------------------------
-----"<<endl;

  SetT();
  TimingFit();
}

void InoTrackFitAlg::TimingFit() { //( CandFitTrackCamHandle &cth)
  if(debug_fit) { cout <<" InoTrackFitAlg : TimingFit" << endl;}

  // Initialisations
  double ss;
  double ts; double qr;  int nc=0;
  double MinUncertainty = 0.; double MinCT=-3000.;

  // Time of first strip in track
  StripListTime=9.e10;

  // Create an offset such that dS=0 at the MinPlane
  double dSOffset=0.; double Sign=-1.; double dS[doubleLa]; // GMA need to put f
rom db
  if(ZIncreasesWithTime==true) {
    dSOffset=fTrackCand->GetdS(MinPlane);
    Sign=1.;
  }
  //asm_170311

  // Store data needed in arrays. Pulse is in PEs.
  double Qp[doubleLa];  double Qm[doubleLa];
  double CTp[doubleLa]; double CTm[doubleLa];
  int Skipp[doubleLa];  int Skipm[doubleLa];
  double C=3.e8;

  double ErrorParam[3];
  ErrorParam[0]=0.; ErrorParam[1]=0.; ErrorParam[2]=0.;

  // Zero the arrays
  for(unsigned int ij=0; ij<nLayer; ++ij) {
    dS[ij]=0.; Qp[ij]=0.; Qm[ij]=0.; CTp[ij]=0.;
    CTm[ij]=0.; Skipp[ij]=0; Skipm[ij]=0;
  }

  // Organise timing for the Far Detector

  // Parameters for PE vs time fit residual
  //GMA all these number need to be updated for INO
  MinUncertainty=0.56;
  ErrorParam[0]=0.56; ErrorParam[1]=0.50; ErrorParam[2]=-0.34;

  // cout <<" fInoTrackFitAlg : TimingFit" << endl;
  // Loop over all planes
  for(int ij=MinPlane; ij<=MaxPlane; ++ij) {
```



```
    if(InitTrkClustData[ij].size()>0) {
      dS[ij]=Sign*(dSOffset-fTrackCand->GetdS(ij));
      CTp[ij]=C*fTrackCand->GetT(ij); //,StripEnd::kPositive);
      // CTm[ij]=C*fTrackCand->GetT(ij,StripEnd::kNegative);

      if(CTp[ij]>MinCT && CTp[ij]<StripListTime) {StripListTime=CTp[ij];}
      // if(CTm[ij]>MinCT && CTm[ij]<StripListTime) {StripListTime=CTm[ij];}
      for(unsigned int jk=0; jk<InitTrkClustData[ij].size(); ++jk) {
        Qp[ij]+=InitTrkClustData[ij][jk].csh->GetPulse(); //StripEnd::kPositive)
;
        // Qm[ij]+=InitTrkClustData[ij][jk].csh->GetPulse(StripEnd::kNegative);
      }
    }
  }
  // Subtract StripList time
  if(StripListTime<8.e30) {
    for(int ij=MinPlane; ij<=MaxPlane; ++ij) {
      if(InitTrkClustData[ij].size()>0) {
        CTp[ij]-=StripListTime;
        // CTm[ij]-=StripListTime;
      }
    }
  } else {
    StripListTime=0.;
  }
  // cout <<" eInoTrackFitAlg : TimingFit" << endl;
  // Carry out a simple straight line fit for T vs dS
  // Sqt: sum of charge*time, Sqss: sum of charge*dS*dS, etc.
  double Sqs=0; double Sqt=0; double Sqss=0; double Sqst=0; double Sqtt=0; doubl
e Sq=0;
  double TimeSlope=-999; double TimeOffset=-999; double RMS=-999;
  double CTCut = 0.; bool CalculateChi2=true;

  // On first iteration, carry out simple fit. Remove outlying points on subsequ
ent passes.
  for(int itr=0; itr<3; ++itr) {
    for(int ij=MinPlane; ij<=MaxPlane; ++ij) {
      // Only consider planes where we found our final strips
      if(InitTrkClustData[ij].size()>0) {
        // For positive strip ends
        ss=dS[ij]; qr=Qp[ij]; ts=CTp[ij];

        if(qr>0. && ts>MinCT && Skipp[ij]==0) {
          if(itr==0) {
            Sq+=qr; Sqs+=qr*ss; Sqt+=qr*ts; Sqss+=qr*ss*ss; Sqst+=qr*ss*ts; Sqtt
+=qr*ts*ts; nc++;
          } else if(fabs(ts-TimeOffset-(ss*TimeSlope)) > CTCut) {
            Sqs-=qr*ss; Sqt-=qr*ts; Sqss-=qr*ss*ss; Sqst-=qr*ss*ts; Sqtt-=qr*ts*
ts; Sq-=qr; nc--; Skipp[ij]=1;
          }
        }
      }
    }

    // Calculate parameters
    if( (Sq*Sqss-Sqs*Sqs)!=0. && Sq!=0.) {
      TimeSlope  = (Sq*Sqst-Sqs*Sqt)/(Sq*Sqss-Sqs*Sqs);
      TimeOffset = (Sqt*Sqss-Sqs*Sqst)/(Sq*Sqss-Sqs*Sqs);
      if( ((Sqtt/Sq)-((Sqt/Sq)*(Sqt/Sq)))>0.) {
        RMS = pow((Sqtt/Sq)-((Sqt/Sq)*(Sqt/Sq)),0.5);
        CTCut = 3.+RMS;
      } else {
        CTCut = 3.5;
      }
    } else {
      CalculateChi2=false;
      break;
    }
  }
```

```
  // cout <<" dInoTrackFitAlg : TimingFit" << endl;

  // Set timing properties for the fitted track
  if(nc!=0 && CalculateChi2==true) {
    // Offset, slope and vtx/end times
    fTrackCand->SetTimeOffset((TimeOffset+StripListTime)/C);
    fTrackCand->SetTimeSlope(TimeSlope/C);

    if(ZIncreasesWithTime==true) {
      fTrackCand->SetVtxT((TimeOffset+StripListTime)/C);
      fTrackCand->SetEndT((TimeOffset+StripListTime)/C+(dS[MaxPlane]*TimeSlope/C
));
    } else {
      fTrackCand->SetEndT((TimeOffset+StripListTime)/C);
      fTrackCand->SetVtxT((TimeOffset+StripListTime)/C+(dS[MaxPlane]*TimeSlope/C
));
    }
    // cout <<" cInoTrackFitAlg : TimingFit" << endl;
    // Chi2
    double Uncertainty; double Residual2; double Chi2=0;

    for(int ij=MinPlane; ij<=MaxPlane; ++ij) {
      if(InitTrkClustData[ij].size()>0) {
        // For positive strip ends
        ss=dS[ij]; qr=Qp[ij]; ts=CTp[ij];
        if(qr>0. && ts>MinCT && Skipp[ij]==0) {
          Residual2=pow(ts-TimeOffset-(ss*TimeSlope),2);
          // From a rough parameterisation of uncertainty (in CT) vs number of P
Es
          if (qr<20) {
            Uncertainty = ErrorParam[0]+exp(ErrorParam[1]+ErrorParam[2]*qr);
          } else {
            Uncertainty=MinUncertainty;
          }
          if(Uncertainty!=0.) {
            Chi2+=Residual2/pow(Uncertainty,2);
          }
        }
      }
    }
    // Set these properties
    fTrackCand->SetTimeFitChi2(Chi2);
    fTrackCand->SetNTimeFitDigit(nc);
  }

  // Now carry out fits with gradients constrained to be +/- c
  double CTIntercept[2]; double Csigma[2]; double Ctrunc[2];
  double ChiSqPositive=-999; double ChiSqNegative=-999;
  int ChiSqNdfPos=-999; int ChiSqNdfNeg=-999;
  double Swtt[2]; double Swt[2]; double Sw[2]; int npts[2]={0,0};
  // cout <<" bInoTrackFitAlg : TimingFit" << endl;
  if(Sq!=0.){
    CTIntercept[0]=Sqt/Sq; Csigma[0]=-99999.9; Ctrunc[0]=-99999.9;
    CTIntercept[1]=Sqt/Sq; Csigma[1]=-99999.9; Ctrunc[1]=-99999.9;

    for(int itr=0; itr<2; ++itr) {
      Swtt[0]=0.; Swt[0]=0.; Sw[0]=0.; npts[0]=0;
      Swtt[1]=0.; Swt[1]=0.; Sw[1]=0.; npts[1]=0;

      for(unsigned int ij=0; ij<nLayer; ++ij) {
        // For positive strip ends
        if(Qp[ij]>0. && CTp[ij]>MinCT) {
          qr=Qp[ij];
          ts=CTp[ij]-dS[ij]+CTIntercept[0];
          if(Ctrunc[0]<0. || fabs(ts)<Ctrunc[0]) {Swtt[0]+=qr*ts*ts; Swt[0]+=qr*
ts; Sw[0]+=qr; ++npts[0];}

          ts=CTp[ij]+dS[ij]+CTIntercept[1];
```



```
          if(Ctrunc[1]<0. || fabs(ts)<Ctrunc[1]) {Swtt[1]+=qr*ts*ts; Swt[1]+=qr*
ts; Sw[1]+=qr; ++npts[1];}
        }
      }

      // Results for fit with gradient +C
      if(npts[0]>1 && Sw[0]!=0.) {
        CTIntercept[0]=CTIntercept[0]-Swt[0]/Sw[0]; Csigma[0]=0.;
        if((Swtt[0]/Sw[0])-(Swt[0]/Sw[0])*(Swt[0]/Sw[0])>0.) {Csigma[0]=pow((Swt
t[0]/Sw[0])-(Swt[0]/Sw[0])*(Swt[0]/Sw[0]),0.5);}
        ChiSqPositive=Csigma[0]; ChiSqNdfPos=npts[0]-1;
        Ctrunc[0]=Csigma[0]+3.;
      }

      // Results for fit with gradient -C
      if(npts[1]>1 && Sw[1]!=0.) {
        CTIntercept[1]=CTIntercept[1]-Swt[1]/Sw[1]; Csigma[1]=0.;
        if((Swtt[1]/Sw[1])-(Swt[1]/Sw[1])*(Swt[1]/Sw[1])>0.) {Csigma[1]=pow((Swt
t[1]/Sw[1])-(Swt[1]/Sw[1])*(Swt[1]/Sw[1]),0.5);}
        ChiSqNegative=Csigma[1]; ChiSqNdfNeg=npts[1]-1;
        Ctrunc[1]=Csigma[1]+3.;
      }
    }
  }
  // Set these properties
  fTrackCand->SetTimeForwardFitRMS(ChiSqPositive);
  fTrackCand->SetTimeForwardFitNDOF(ChiSqNdfPos);
  fTrackCand->SetTimeBackwardFitRMS(ChiSqNegative);
  fTrackCand->SetTimeBackwardFitNDOF(ChiSqNdfNeg);

  // cout <<" aInoTrackFitAlg : TimingFit" << ChiSqPositive <<" "<<ChiSqNdfPos<<
" "<<ChiSqNegative<<" "<<ChiSqNdfNeg<< endl;
}

/*
  void InoTrackFitAlg::SetRangeAnddS( ) //CandFitTrackCamHandle& cth)
  {
  // Set range and dS as calculated by the swimmer
  // cout <<" InoTrackFitAlg : SetRangeAnddS from swimmer values " << endl;

  // int ZDir;
  // int VtxPlane=5000; int EndPlane=-20; int Increment=1;
  int VtxPlane=5000; int EndPlane=0; int Increment=1;
  double dS; double dRange; double dP;

  // Start at the end of the track and calculate the required additions to range

  // find ending Z position (defined as Z position where muon has 100 MeV of res
idual energy.  This corresponds to 1/2 inch of Fe.

  // NOTE: Average dP for 1" iron is 95 MeV.

  //if(ZIncreasesWithTime==true) {ZDir=1; EndPlane=MaxPlane; VtxPlane=MinPlane;
Increment=-1;}
  //else {ZDir=-1; EndPlane=MinPlane; VtxPlane=MaxPlane; Increment=1;}

  //PlexPlaneId plnid(detector,EndPlane,false);
  //PlexPlaneId endplnid(detector,EndPlane,false);

  //double Zscint = ZPosLayer[EndPlane]; //SlcClustData[EndPlane][0].csh->GetZPo
s();
  //double Znextscint = Zscint;

  //UgliScintPlnHandle scintpln = ugh.GetScintPlnHandle(plnid);
  //double Zend = Zscint + double(ZDir)*scintpln.GetHalfThickness();

  //PlexPlaneId nextscint =  endplnid.GetAdjoinScint(ZDir);
  //UgliScintPlnHandle nextscintpln = ugh.GetScintPlnHandle(nextscint);
  //if(nextscintpln.IsValid() && nextscint.GetPlaneView()!=PlaneView::kUnknown){
```

```
  //Znextscint = nextscintpln.GetZ0();
  //}
  //else
  //{
  //nextscint = endplnid;
  //}

  //plnid = plnid.GetAdjoinSteel(ZDir);
  //if(plnid.IsValid()){
  //UgliSteelPlnHandle steelpln = ugh.GetSteelPlnHandle(plnid);
  //Zend = steelpln.GetZ0() - double(ZDir)*steelpln.GetHalfThickness();
  //}

  // add two planes of steel for the ND spectrometer
  //if(detector==Detector::kNear && EndPlane>=121) {
  //for(int i=0;i<2;i++){
  //if(plnid.GetAdjoinSteel(ZDir).IsValid()){
  //    PlexPlaneId plnid_after = plnid.GetAdjoinSteel(ZDir);
  //    if(plnid_after.IsValid()) {
  //    plnid = plnid_after;
  //    UgliSteelPlnHandle steelpln = ugh.GetSteelPlnHandle(plnid);
  //    Zend = steelpln.GetZ0() - double(ZDir)*steelpln.GetHalfThickness();
  //    }
  //    }
  //    }
  //}


  // GMA This is to arbitray value just to have compilation
  // Need to put the RPC/IRON postion properly
  double Zend = LayerThickness*(EndPlane-1);

  //  double angle = 0;
  //  double ct = cos(angle);
  //  double st = sin(angle);
  // Determine whether track stops in coil
  //  float u_end = FilteredData[EndPlane][0].x_k0;
  //  float v_end = FilteredData[EndPlane][0].x_k1;
  //  float du_end = FilteredData[EndPlane][0].x_k2;
  //  float dv_end = FilteredData[EndPlane][0].x_k3;
  //  float delz = 0.0852; // Znextscint-Zscint;
  //  float u_extrap = u_end +delz*du_end;
  //  float v_extrap = v_end +delz*dv_end;
  //  float x_extrap = (ct*u_extrap-st*v_extrap);
  //  float y_extrap = (st*u_extrap+ct*v_extrap);


  //PlaneCoverage::PlaneCoverage_t kPC = PlaneCoverage::kComplete;
  //if(detector==Detector::kNear) kPC=PlaneCoverage::kNearFull;

  //bool isInOutline = fPL.IsInside(x_extrap,y_extrap,nextscint.GetPlaneView(),k
PC,false);
  //bool isInCoil = isInOutline && !fPL.IsInside(x_extrap,y_extrap,nextscint.Get
PlaneView(),kPC,true);


  double S = 0; double Range = 0; double Prange = 0.0;
  double StateVector[6]; double Output[6];
  double chargesign = -1;
  bool GoForward = true; bool done=true; bool swimOK=true;

  // if in coil find midpoint and swim towards last clust from there

  //    if(isInCoil){
  //    float zCoil = Znextscint;
  //    float u_extrapC = u_extrap;
  //    float v_extrapC = v_extrap;
  //    float x_extrapC = x_extrap;
  //    float y_extrapC = y_extrap;
```



```
  //    while(isInCoil){
  //    zCoil -= 1.0*Munits::cm*ZDir;
  //    float delzC = zCoil - Zscint;
  //    u_extrapC = u_end +delzC*du_end;
  //    v_extrapC = v_end +delzC*dv_end;
  //    x_extrapC = (ct*u_extrapC-st*v_extrapC);
  //    y_extrapC = (st*u_extrapC+ct*v_extrapC);
  //    // isInCoil = !fPL.IsInside(x_extrapC,y_extrapC,nextscint.GetPlaneView()
,kPC,true);
  //    }
  //    float zMinCoil = zCoil;
  //    if(zMinCoil<Zscint && ZDir==1)  zMinCoil=Zscint;
  //    if(zMinCoil>Zscint && ZDir==-1) zMinCoil=Zscint;

  //    zCoil = Znextscint;

  //    isInCoil = true;
  //    while(isInCoil){
  //    zCoil += 1.0*Munits::cm*ZDir;
  //    float delzC = zCoil - Zscint;
  //    u_extrapC = u_end +delzC*du_end;
  //    v_extrapC = v_end +delzC*dv_end;
  //    x_extrapC = (ct*u_extrapC-st*v_extrapC);
  //    y_extrapC = (st*u_extrapC+ct*v_extrapC);
  //    isInCoil = !fPL.IsInside(x_extrapC,y_extrapC,nextscint.GetPlaneView(),kP
C,true);
  //    }


  // float zMaxCoil = zCoil;
  // float zmin; float zmax;
  // ugh.GetZExtent(zmin,zmax);
  // if(zMaxCoil>zmax && ZDir==1)  zMaxCoil=zmax;
  // if(zMaxCoil<zmin && ZDir==-1) zMaxCoil=zmin;

  // now swim from mid-coil back to endplane
  // float zMidCoil = 0.5*(zMinCoil + zMaxCoil);
  // float delzC  = zMidCoil -Zscint;
  // u_extrapC = u_end +delzC*du_end;
  // v_extrapC = v_end +delzC*dv_end;
  // x_extrapC = 0.707*(u_extrapC-v_extrapC);
  // y_extrapC = 0.707*(u_extrapC+v_extrapC);

  // StateVector[0] = u_extrapC; Output[0]=StateVector[0];
  // StateVector[1] = v_extrapC; Output[1]=StateVector[1];
  // StateVector[2] = FilteredData[EndPlane][0].x_k2; Output[2]=StateVector[2];
  // StateVector[3] = FilteredData[EndPlane][0].x_k3; Output[3]=StateVector[3];
  // chargesign = -1;
  // if(FilteredData[EndPlane][0].x_k4!=0.) {chargesign =  FilteredData[EndPlane
][0].x_k4/fabs(FilteredData[EndPlane][0].x_k4);}

  // GoForward = !ZIncreasesWithTime;
  // StateVector[4] = 10.*chargesign; Output[4]=StateVector[4];
  // StateVector[5] = FilteredData[EndPlane][0].x_k5; Output[5]=StateVector[5];

  // double dsdz = pow((1. + pow(StateVector[2],2) + pow(StateVector[3],2)),0.5)
;
  // set fallback to nominal energy loss in case coil swim fails
  // Prange = 0.095*dsdz;
  // if(detector==Detector::kNear && EndPlane>121) Prange = 0.2*dsdz;
  // Prange += 0.5*dsdz*0.1*fabs(zMaxCoil-zMinCoil)*2.357*1.97;

  // swimOK = Swim(StateVector, Output, zMidCoil, EndPlane , GoForward, &dS, &dR
ange, &dP);

  // if(swimOK ){
  // S = dS; Range = dRange; Prange = fabs(dP);
  // fTrackCand->SetdS(EndPlane,S);
  // fTrackCand->SetRange(EndPlane,Range);
```

```
  // }
  // if(!swimOK) {Output[4] = chargesign/Prange;}
  // } else


  {
  // // normal case - track does not end in coil
  //    if((Zend<Zscint && ZDir==1) || (Zend>Zscint && ZDir==-1)) {
  //      cout <<" InoTrackFitAlg :  Zend on wrong side of last scint! " << endl
;
  //      Zend=Zscint;
  //    }

  // now swim to Zend
  StateVector[0]=FilteredData[EndPlane][0].x_k0; Output[0]=StateVector[0];
  StateVector[1]=FilteredData[EndPlane][0].x_k1; Output[1]=StateVector[1];
  StateVector[2]=FilteredData[EndPlane][0].x_k2; Output[2]=StateVector[2];
  StateVector[3]=FilteredData[EndPlane][0].x_k3; Output[3]=StateVector[3];
  StateVector[4]=FilteredData[EndPlane][0].x_k4; Output[4]=StateVector[4];
  StateVector[5]=FilteredData[EndPlane][0].x_k5; Output[5]=StateVector[5];

  chargesign = -1;
  if(StateVector[4]!=0.) {chargesign = StateVector[4]/fabs(StateVector[4]);}

  GoForward = ZIncreasesWithTime;
  done = Swim(StateVector, Output, EndPlane, Zend, GoForward,  &dS, &dRange, &dP
);

  GoForward = !ZIncreasesWithTime;
  double dsdz = pow((1. + pow(StateVector[2],2) + pow(StateVector[3],2)),0.5);
  S = 0; Range = 10.0*dsdz; Prange = 0.095*dsdz;
  swimOK = false;
  if(done){
  for(int j=0;j<6;j++) {StateVector[j]=Output[j];}

  // now swim from Zend to EndPlane
  StateVector[4] = chargesign * 10.52;  // start @ P = 100 MeV (Eloss in 1/2 " I
ron)
  swimOK = Swim(StateVector, Output, Zend, EndPlane , GoForward, &dS, &dRange, &
dP);
  if(swimOK){
  S += dS; Range += dRange; Prange += fabs(dP);
  fTrackCand->SetdS(EndPlane,S);
  fTrackCand->SetRange(EndPlane,Range);
  }
  }
  if(!swimOK) {Output[4] = chargesign/Prange;}
  }

  int thisplane = EndPlane;
  // now swim back to vertex
  bool firstplane=true;
  for(int i=EndPlane+Increment; Increment*i<=Increment*VtxPlane; i+=Increment) {
  if(FilteredData[i].size()>0) {
  double delU = FilteredData[i][0].x_k0 - StateVector[0] ;
  double delV = FilteredData[i][0].x_k1 - StateVector[1] ;
  double dSperPlane=0.;
  if(thisplane!=i) {dSperPlane = pow(delU*delU + delV*delV,0.5)/double(abs(thisp
lane-i));}

  // only update state vector if change in U/V is reasonable.
  if(dSperPlane < 1.5) { // *TMinuit::m) {
  StateVector[0]=FilteredData[i][0].x_k0;
  StateVector[1]=FilteredData[i][0].x_k1;
  StateVector[2]=FilteredData[i][0].x_k2;
  StateVector[3]=FilteredData[i][0].x_k3;

  chargesign=-1;
  if(FilteredData[i][0].x_k4!=0.) {chargesign = FilteredData[i][0].x_k4/fabs(Fil
```



```
teredData[i][0].x_k4);}

  StateVector[5]=FilteredData[i][0].x_k5;
  }

  StateVector[4] = chargesign * fabs(Output[4]);
  done = Swim(StateVector, Output, thisplane, i , GoForward, &dS, &dRange, &dP);
  if(done){
  S+=dS; Range+=dRange; Prange+=fabs(dP);
  fTrackCand->SetdS(i,S); fTrackCand->SetRange(i,Range);
  firstplane=false;
  }
  else {
  cout <<" InoTrackFitAlg :  swim fail " << endl;
  }
  thisplane=i;
  }
  }


  //  PlexPlaneId vtxplnid(detector,VtxPlane,false);
  //  PlexPlaneId plnid_before = vtxplnid.GetAdjoinSteel(-ZDir);

  //if(plnid_before.IsValid()) {
  //  plnid = plnid_before;
  //  UgliSteelPlnHandle steelpln = ugh.GetSteelPlnHandle(plnid);
  //  double Zstart = steelpln.GetZ0();
  //  StateVector[0]=FilteredData[VtxPlane][0].x_k0;
  //  StateVector[1]=FilteredData[VtxPlane][0].x_k1;
  //  StateVector[2]=FilteredData[VtxPlane][0].x_k2;
  //  StateVector[3]=FilteredData[VtxPlane][0].x_k3;
  //  StateVector[4]=Output[4];
  //  StateVector[5]=FilteredData[VtxPlane][0].x_k5;

  //  Swim(StateVector, Output, VtxPlane, Zstart, GoForward, &dS,&dRange,&dP);
  //  S+=dS; Range+=dRange; Prange+=fabs(dP);

  //  track->SetRange(VtxPlane,Range);
  //  track->SetdS(VtxPlane,S);
  //}


  // if Prange < 21 GeV, use this value.  Otherwise, use finder track energy, wh
ich is somewhat less prone to gross errors.

  // apply fudge factor for nominal steel thickness in ND geometry.

  //  track->SetMomentumRange(Prange*ecorr);
  //  CandTrackHandle* findertrack = track->GetFinderTrack();
  //  if(((detector==Detector::kFar && Prange>21.) || (detector==Detector::kNear
 && Prange>12.)) && findertrack) {track->SetMomentumRange(findertrack->GetMoment
um());}

  }
*/

/*
  void InoTrackFitAlg::SetPropertiesFromFinderTrack(CandFitTrackCamHandle &cthx)
  {
  // This method is only called if the fit fails. We set properties from finder
track.
  // This clearly does not include fitted properties such as q/p or QPVtxError.
  cout <<" InoTrackFitAlg : SetPropertiesFromFinderTrack" << endl;
  cthx.SetDirCosU(track->GetDirCosU());
  cthx.SetDirCosV(track->GetDirCosV());
  cthx.SetDirCosZ(track->GetDirCosZ());
  cthx.SetVtxU(track->GetVtxU());
  cthx.SetVtxV(track->GetVtxV());
  cthx.SetVtxZ(track->GetVtxZ());
```

```
  cthx.SetVtxT(track->GetVtxT());
  cthx.SetVtxPlane(track->GetVtxPlane());

  cthx.SetEndDirCosU(track->GetEndDirCosU());
  cthx.SetEndDirCosV(track->GetEndDirCosV());
  cthx.SetEndDirCosZ(track->GetEndDirCosZ());
  cthx.SetEndU(track->GetEndU());
  cthx.SetEndV(track->GetEndV());
  cthx.SetEndZ(track->GetEndZ());
  cthx.SetEndT(track->GetEndT());
  cthx.SetEndPlane(track->GetEndPlane());

  cthx.SetMomentumRange(track->GetMomentum());
  cthx.SetMomentum(track->GetMomentum());

  cthx.SetTimeSlope(track->GetTimeSlope());
  cthx.SetTimeOffset(track->GetTimeOffset());
  cthx.SetTimeFitChi2(track->GetTimeFitChi2());
  cthx.SetNTimeFitDigit(track->GetNTimeFitDigit());
  cthx.SetTimeForwardFitRMS(track->GetTimeForwardFitRMS());
  cthx.SetTimeForwardFitNDOF(track->GetTimeForwardFitNDOF());
  cthx.SetTimeBackwardFitRMS(track->GetTimeBackwardFitRMS());
  cthx.SetTimeBackwardFitNDOF(track->GetTimeBackwardFitNDOF());


  // Set quantities required at each plane in finder track
  int direction=1;
  if(ZIncreasesWithTime==false) {direction=-1;}

  for(int i=cthx.GetVtxPlane(); i*direction<=cthx.GetEndPlane()*direction; i+=di
rection){
  if(track->IsTPosValid(i)) {
  cthx.SetTrackPointError(i,track->GetTrackPointError(i));
  cthx.SetdS(i,track->GetdS(i));
  cthx.SetRange(i,track->GetRange(i));
  cthx.SetU(i,track->GetU(i));
  cthx.SetV(i,track->GetV(i));
  }
  }

  CalculateTrace(); //cthx);
  SetT(); //&cthx);

  Calibrate(); //&cthx);

  }
*/

void InoTrackFitAlg::Trace(const char * /* c */) const
{;
}
void InoTrackFitAlg::SetT() {
  // we take a weighted average of clusts in the same
  // plane.  the proper way to do this would be to consider clusts coming
  // from the same PMT and use the earliest time.

  //cout <<" inoTrackFitAlg : starting SetT" << endl;
  //for (unsigned ij=0; ij<fFinderTrack->ClustsInTrack.size(); ij++)
  //{
  //}
}
void InoTrackFitAlg::GoForwards(bool first) {
  // The bool variable 'first' denotes first iteration; for 1st half of the 1st
iteration,
  // when KF pocesses hits along increasing z, first = 0. For the 1st half of su
bsequent
  // iterations, first = 1. This is true for up going Î¼. For downgoing Î¼, it i
s always set to 0.
  cout<<"GoForwards_new : "<<"first = "<<first<<endl;
```



```
  double x__minus[5]={0.0};
  bool GoForward = true;
  cout<<"GoForward"<< GoForward<<endl;

  cout<<"MaxPlane"<< MaxPlane<<endl;
  cout<<"MinPlane"<< MinPlane<<endl;
  Int_t StartPlane = MinPlane;
  cout<<"StartPlane"<< StartPlane<<endl;
  cout<<"EndofRangePlane"<< EndofRangePlane<<endl;
  if(!ZIncreasesWithTime) {
    cout<<"ZIncreasesWithTime"<< ZIncreasesWithTime<<endl;
    StartPlane = EndofRangePlane;
    cout<<"StartPlane"<< StartPlane<<endl;
  } else {
    EndofRangePlane = MaxPlane;
  }

  unsigned ntrk = fTrackCand->GetClusterEntries();      // number of hits in a t
rack(let)
  cout<<"ntrk"<< ntrk<<endl;
  cout<<"first"<< first<<endl;
  int iend = (first) ? 0 : 1;                                  // when first =
0, iend = 1 and vice versa

  cout <<"GoForwards_new : "<<"ntrksize = "<<ntrk<<"  "<<"ntrk-iend = "<<ntrk-iend<<endl;
  for (unsigned ijk=0; ijk<ntrk-iend; ijk++) {
    // ijk runs over the (available number of hits - (0 or 1)) in a track(let)
    if (!fTrackCand->ClustsInTrack[ijk]->GetStraight()) {
      continue;
    }
    int ij = fTrackCand->ClustsInTrack[ijk]->GetZPlane();
    // This gives that latest plane where we have experimental data and/or filte
red data

    EndofRange = false;
    // Next we find the next plane and other stuff used in the previous algorith
m
    if (TrkClustsData[ij].size()>0) {
      if (PassTrack) {
        // Find Next Plane
        int NewPlane=-99;

        //if (!first && ijk !=ntrk-1 && fTrackCand->ClustsInTrack[ijk+1]->GetStr
aight())
        if ((ijk !=ntrk-1) && (fTrackCand->ClustsInTrack[ijk+1]->GetStraight()))
 {
          // this is done in the first half of the first iteration
          NewPlane=fTrackCand->ClustsInTrack[ijk+1]->GetZPlane();
        }

        if (NewPlane!=-99) {
          // Define measurement function
          int PlaneView = TrkClustsData[NewPlane][0].PlaneView;
          // GMA for Clusts this condition is fine, but for cluster, this is not
 correct
          for (int jk=0; jk<2; jk++) {
            for (int kl=0; kl<5; kl++) {
              H_k[jk][kl]=0;
            }
          }
          if (PlaneView%2==0) {H_k[0][0]=1;}
          if (PlaneView   >0) {H_k[1][1]=1;}

          ds=0;
          drange=0;
          PredictedStateCov(x_k_minus, ij, NewPlane, GoForward, x__minus, 0, &ds
, &drange);
          CalcKalmanGain(x__minus, NewPlane);
          KalmanFilterStateVector(x__minus, NewPlane, GoForward,x_k);
```

```
          UpdateCovMatrix(NewPlane);
          MoveArrays(NewPlane,GoForward);
          if(SaveData) {StoreFilteredData(NewPlane);}
          //if (ZIncreasesWithTime && LastIteration && (NewPlane==MaxPlane))
          //cout<<"reached end: P = "<<1./x_k[4]<<endl;
        }
        // bracket controlling legitimacy of NewPlane
      }
      // bracket checking if PassTrack is true
    }
    // bracket checking if no. of clusters > 0. // (TrackClusterDataSize >0 ends
)

    // JAM end of range found
    if(EndofRange && LastIteration && ZIncreasesWithTime) {
      cout<<"This is redundant – never used"<<endl;
      EndofRangePlane=ij;
      break;
    }
  }
  // end of 'for' loop over available hits

  // Store entries from covariance matrix for use in setting track properties.
  if(LastIteration) {
    if(ZIncreasesWithTime==true) {
      //cout<<"âM-^FM-^Q GF error@End"<<endl;
      EndCov[0]=C_k[0][0]; EndCov[1]=C_k[1][1];
      EndCov[2]=C_k[2][2]; EndCov[3]=C_k[3][3];
      EndCov[4]=C_k[4][4];

      EndState[0]=x_k[0];                EndState[1]=x_k[1];
      EndState[2]=x_k[2];                EndState[3]=x_k[3];
      EndState[4]=x_k[4];
    } else {
      //cout<<"âM-^FM-^S GF error@Vtx"<<endl;
      VtxCov[0]=C_k[0][0]; VtxCov[1]=C_k[1][1];
      VtxCov[2]=C_k[2][2]; VtxCov[3]=C_k[3][3];
      VtxCov[4]=C_k[4][4];
    }
  }
}

void InoTrackFitAlg::GoBackwards(bool first) {
  cout<<"GoBackwards_new : "<<"first = "<<first<<endl;

        // Carry out the Kalman fit along the track in the direction of decreasi
ng z
  cout <<" InoTrackFitAlg : GoBackwards_new, carry out fit in negative z direction" << endl;

  double x__minus[5] = {0.0};
  bool GoBackward = false;

  Int_t StartPlane = MaxPlane;// Int_t EndPlane=MinPlane;


  cout<<"GoBackwards_new : "<<"MinPlane = "<<MinPlane<<"   "<<"EndofRangePlane = "<<EndofRa
ngePlane<<"   "<<"MaxPlane = "<<MaxPlane<<"StartPlane "<<StartPlane<<" EndofRange "<<Endo
fRangePlane<<endl;

  if(ZIncreasesWithTime) {
    StartPlane = EndofRangePlane;
  } else {
    EndofRangePlane = MinPlane;
  }
  cout<<"GoBackwards_new : "<<"MinPlane = "<<MinPlane<<"   "<<"EndofRangePlane = "<<EndofRa
ngePlane<<"   "<<"MaxPlane = "<<MaxPlane<<"StartPlane "<<StartPlane<<" EndofRange "<<Endo
fRangePlane<<endl;

  int ntrk = fTrackCand->GetClusterEntries();
```



```
  int iend = (first) ? 0 : 1;
  cout<<"GoBackwards_new : "<<"ntrksize = "<<ntrk<<"   "<<"iend = "<<iend<<endl;

  for (int ijk=ntrk-1; ijk>=iend; ijk--) {
    cout<<"ijk "<<ijk<<endl;
    if (!fTrackCand->ClustsInTrack[ijk]->GetStraight()) {
      cout<<"ijk "<<ijk<<endl;
      continue;
    }
    int ij = fTrackCand->ClustsInTrack[ijk]->GetZPlane();

    if (TrkClustsData[ij].size()>0) {
      cout<<"GoBackwards_new : "<<"PassTrack = "<<PassTrack<<endl;

      if (PassTrack) {
        //Find Prev Plane
        int NewPlane=-99;            // int jk=(ij-1);
        cout<<"NewPlane_ "<<NewPlane<<endl;
        //for (int jk=0; jk<6; jk++) {x_k_old[jk] = x_k_minus[jk];}

        //a     cout <<"backfirst "<<int(first)<<" "<<ijk<<" "<<ntrk-1<<endl;
        //if (!first && ijk !=0 && fTrackCand->ClustsInTrack[ijk-1]->GetStraight
())
        if (ijk !=0 && fTrackCand->ClustsInTrack[ijk-1]->GetStraight()) {
          NewPlane=fTrackCand->ClustsInTrack[ijk-1]->GetZPlane();
          cout<<"NewPlane "<<NewPlane<<endl;
        }
        //*****************************************************
        /*
          else {
          if (fTrackCand->GetEntries()>=MINLAYER) {
          int plane = ij;
          // int loopmx = ((ijk<ntrk-1) && (ijk>0)) ? ij - MinPlane : 3;
          int loopmx = ((ijk>0)) ? max(ij-MinPlane, 5) : 4;
          double dsBefore=0;
          double drangeBefore=0;
          /////////////////////////////////////////////////
          for (int nloop=0; nloop <loopmx; nloop++) {
          // nloop : deals with cases with considerable gap between successive h
its; i.e. the hits are
          // not in the consecutive planes.
          if (nloop==0) {
          //                                                            cout<<"n
loop = 0 here; so x_k_minus is fed to StateVector"<<endl;
          for(int kl=0; kl<6; ++kl) {
          StateVector[kl]=x_k_minus[kl];
          }
          } else {
          //                                                            cout<<"n
loop != 0 here; Prediction is fed to StateVector"<<endl;
          for(int kl=0; kl<6; ++kl) {
          StateVector[kl]= x__minus[kl];
          }
          }
          double ds=0;
          double drange=0;
          int nextplane = -99;

          bool GetPrediction=PredictedStateCov(StateVector, plane, nextplane, fa
lse, Prediction, 0); //, &ds, &drange);

          if (!GetPrediction || nextplane<0 || nextplane>=int(nLayer)) break;

          if (GetPrediction) {
          if (nextplane > plane) {
          if (TrkClustsData[nextplane+shiftLa].size()>0) {
          StoreFilteredData_sr(nextplane+shiftLa, x__minus, false);
          fTrackCand->f2dS[nextplane+shiftLa] =ds+dsBefore;
          fTrackCand->f2Range[nextplane+shiftLa] =drange+drangeBefore;
```

```
          dsBefore = drangeBefore = 0;
          //
cout<<"GB"<<"    "<<"HiHi"<<endl;
          } else {
          StoreFilteredData_sr(nextplane+shiftLa, x__minus, false);
          dsBefore +=ds;
          drangeBefore +=drange;
          //
cout<<"GB"<<"    "<<"HaHa"<<endl;
          }
          } else {
          plane = nextplane;
          if (TrkClustsData[plane].size() >0) {
          //
cout<<"plane= "<<plane<<endl;
          NewPlane= plane;
          break;
          }
          //StoreFilteredData_sr(nextplane, Prediction, true);
          StoreFilteredData_sr(nextplane, x__minus, true);
          //
cout<<"GB"<<"    "<<"HoHo"<<endl;
          }
          }

          }
          /////////////////////////////////////
          }
          }
        */

        if (NewPlane!=-99) {
          //                                    cout<<"GB: so new plane value ha
s been obtained; NewPlane = "<<NewPlane<<endl;
          // Define measurement function
          int PlaneView = TrkClustsData[NewPlane][0].PlaneView;

          //GMA for Clusts this condition is fine, but for cluster, this is not
correct
          // Carry out the Kalman fit
          for (int jk=0; jk<2; jk++) {
            for (int kl=0; kl<5; kl++) {
              H_k[jk][kl]=0;
            }
          }
          if (PlaneView%2==0) {H_k[0][0]=1;}
          if (PlaneView   >0) {H_k[1][1]=1;}
          ds = 0.0;
          drange = 0.0;
          PredictedStateCov(x_k_minus, ij ,NewPlane, GoBackward, x__minus, 0, &d
s, &drange);
          CalcKalmanGain(x__minus, NewPlane);
          KalmanFilterStateVector(x__minus, NewPlane, GoBackward,x_k);
          UpdateCovMatrix(NewPlane);
          MoveArrays(NewPlane,GoBackward);
          if(LastIteration) {
            GPL += ds;
            RNG += drange;
          }
          if(SaveData) {StoreFilteredData(NewPlane);}
          //if (!ZIncreasesWithTime && LastIteration && (NewPlane==MinPlane))
          //cout<<"reached end: P = "<<1./x_k[4]<<endl;
        }
      }
      //else {cout <<" InoTrackFitAlg : GoBackwards, Outside detector - track fa
iled" << endl;}
    }
    //JAM end of range found
    if(EndofRange && LastIteration && !ZIncreasesWithTime) {
```



```
      EndofRangePlane = ij;
      cout<<"ij "<<ij<<endl;
      break;
    }
  }

  // Store entries from covariance matrix for use in setting track properties
  if(LastIteration) {
    if(ZIncreasesWithTime==true) {
      //cout<<"âM-^FM-^Q GB error@Vtx"<<endl;
      VtxCov[0]=C_k[0][0]; VtxCov[1]=C_k[1][1];
      VtxCov[2]=C_k[2][2]; VtxCov[3]=C_k[3][3];
      VtxCov[4]=C_k[4][4];
    } else {
      //cout<<"âM-^FM-^S GB error@End"<<endl;
      EndCov[0]=C_k[0][0]; EndCov[1]=C_k[1][1];
      EndCov[2]=C_k[2][2]; EndCov[3]=C_k[3][3];
      EndCov[4]=C_k[4][4];

      EndState[0]=x_k[0];                EndState[1]=x_k[1];
      EndState[2]=x_k[2];                EndState[3]=x_k[3];
      EndState[4]=x_k[4];
    }
  }
}


bool InoTrackFitAlg::DirectionFromFinderHitsOldFunc(InoTrack *trk,  double& Find
erPathLength, double& FinderDistance) {
  unsigned int nhits = trk->GetEntries();

  //  double ti         = 0.0;
  double sum_iT = 0.0;
  double sum_iZ = 0.0;
  double sum_iTZ        = 0.0;
  double sum_i2T        = 0.0;

  unsigned Nhit = nhits-1;

  for(unsigned ij = 0; ij < Nhit; ij++) {
    sum_iT      += trk->ClustsInTrack[ij]->GetTime();
    sum_iZ      += trk->ClustsInTrack[ij]->GetZPos();
    sum_iTZ     += trk->ClustsInTrack[ij]->GetTime()*trk->ClustsInTrack[ij]->Get
ZPos();
    sum_i2T += trk->ClustsInTrack[ij]->GetTime()*trk->ClustsInTrack[ij]->GetTime
();
  }
  double dZdT = 0.0;
  dZdT = (sum_iTZ - (sum_iT * sum_iZ)/Nhit)/(sum_i2T-(sum_iT*sum_iT)/Nhit);

  for(unsigned jk = 0; jk < nhits; jk++) {
    if (jk>0) {
      FinderPathLength +=pow(pow(trk->ClustsInTrack[jk]->GetXPos() - trk->Clusts
InTrack[jk-1]->GetXPos(), 2.) +
                             pow(trk->ClustsInTrack[jk]->GetYPos() - trk->Clusts
InTrack[jk-1]->GetYPos(), 2.) +
                             pow(trk->ClustsInTrack[jk]->GetZPos() - trk->Clusts
InTrack[jk-1]->GetZPos(), 2.), 0.5);
    }
  }
  FinderDistance = sqrt((trk->ClustsInTrack[nhits-1]->GetXPos() - trk->ClustsInT
rack[0]->GetXPos())*(trk->ClustsInTrack[nhits-1]->GetXPos() - trk->ClustsInTrack
[0]->GetXPos()) + (trk->ClustsInTrack[nhits-1]->GetYPos() - trk->ClustsInTrack[0
]->GetYPos())*(trk->ClustsInTrack[nhits-1]->GetYPos() - trk->ClustsInTrack[0]->G
etYPos()) + (trk->ClustsInTrack[nhits-1]->GetZPos() - trk->ClustsInTrack[0]->Get
ZPos())*(trk->ClustsInTrack[nhits-1]->GetZPos() - trk->ClustsInTrack[0]->GetZPos
()));

  return (dZdT>0.0) ? true : false;
```

```
}

bool InoTrackFitAlg::DirectionFromFinderHits(InoTrack *trk,  double& FinderPathL
ength, double& FinderDistance) {
  cout<< " InoTrackFitAlg::DirectionFromFinderHits " <<endl;
  int nhits = trk->GetEntries();

  double szxy=0, sz=0, sxy=0, sn=0, sz2=0;
  //  int nused = 0;
  double errsq=TimeError*TimeError;
  for(int ij = 0; ij < nhits; ij++) {
    // cout<<"ZPos = "<<trk->ClustsInTrack[ij]->GetZPos()<<", Time = "<<trk->Clu
stsInTrack[ij]->GetTime()<<endl;
    // for (unsigned int ix=0; ix<trk->ClustsInTrack[ij]->HitsInCluster.size();
ix++) {
    //   cout<<" ix "<<ij<<" "<< ix<<" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->GetZPos()<<" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->GetTime()<<" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetTime()<<" "
    //  //      <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetTrueTim
e()<<" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetXTime()<<" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetXTimeCorr()<<
" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetXTrueTime()<<
" "

    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetYTime()<<" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetYTimeCorr()<<
" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetYTrueTime()<<
" "

    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetXStripNum()<<
" "
    //    <<setw(6)<<trk->ClustsInTrack[ij]->HitsInCluster[ix]->GetYStripNum()<<
" "
    //    <<endl;
    // }
    szxy +=trk->ClustsInTrack[ij]->GetZPos()*trk->ClustsInTrack[ij]->GetTime()/e
rrsq;
    sz +=trk->ClustsInTrack[ij]->GetZPos()/errsq;
    sz2 +=trk->ClustsInTrack[ij]->GetZPos()*trk->ClustsInTrack[ij]->GetZPos()/er
rsq;
    sxy +=trk->ClustsInTrack[ij]->GetTime()/errsq;
    sn +=1/errsq;
    //    nused++;
  }
  double dZdT = 0.0;
  if ((sz2*sn - sz*sz) !=0) {
    dZdT = (szxy*sn - sz*sxy)/(sz2*sn - sz*sz); //(sum_iTZ - (sum_iT * sum_iZ)/n
hits)/(sum_i2T-(sum_iT*sum_iT)/nhits);
  }

  for(int jk = 1; jk < nhits; jk++) {
    FinderPathLength +=pow(pow(trk->ClustsInTrack[jk]->GetXPos() - trk->ClustsIn
Track[jk-1]->GetXPos(), 2.) +
                           pow(trk->ClustsInTrack[jk]->GetYPos() - trk->ClustsIn
Track[jk-1]->GetYPos(), 2.) +
                           pow(trk->ClustsInTrack[jk]->GetZPos() - trk->ClustsIn
Track[jk-1]->GetZPos(), 2.), 0.5);
  }

  FinderDistance = sqrt(
                        (trk->ClustsInTrack[nhits-1]->GetXPos() - trk->ClustsInT
rack[0]->GetXPos())*
                        (trk->ClustsInTrack[nhits-1]->GetXPos() - trk->ClustsInT
rack[0]->GetXPos()) +
```



```
                        (trk->ClustsInTrack[nhits-1]->GetYPos() - trk->ClustsInT
rack[0]->GetYPos())*
                        (trk->ClustsInTrack[nhits-1]->GetYPos() - trk->ClustsInT
rack[0]->GetYPos()) +
                        (trk->ClustsInTrack[nhits-1]->GetZPos() - trk->ClustsInT
rack[0]->GetZPos())*
                        (trk->ClustsInTrack[nhits-1]->GetZPos() - trk->ClustsInT
rack[0]->GetZPos()));

  return (dZdT>0.0) ? true : false;
}

void InoTrackFitAlg::GetPropagator(double *istate, double Bx, double By, double
dBxdx, double dBydx, double dBxdy, double dBydy, double dz, TGeoMaterial* materi
al) {
  for (int ij=0; ij<5; ++ij) {
    for (int jk=0; jk<5; ++jk) {
      F_k_minus[ij][jk]=0;
    }
  }

  double x=0;   double y = 0;   double tx = 0;          double ty = 0;
double qbyP = 0;
  x=istate[0];  y=istate[1];    tx      =istate[2];             ty      =istate[
3];             qbyP=istate[4];

  double        P = 0.0;
  double E = 0.0;
  double b = 0.0;
  double dE= 0.0;
  //  double q = 0.0;
  double d = 0.0;
  double f = 0.0;
  double ag = 0.0;
  double tB= 0.0;

  P     = fabs(1/qbyP);
  if (P >= 100.0000) {
    P = 100.0000000000;                         // This is to make sure that the
 calculation does not blow up in the 1st step
  }
  E = sqrt(P*P + 0.1056*0.1056);
  b = P/E;

  double kappa  = 0.299792458000000;
  double T = sqrt(1 + tx*tx + ty*ty);
  if(ZIncreasesWithTime==false) T=-T;
  double B = sqrt(Bx*Bx + By*By);
  double T2= (1 + tx*tx + ty*ty);
  double h = kappa  *  qbyP  * T;
  //  double RoC = 0.000000000000000;
  if ((Bx != 0.0) && (By != 0.0)) {
    tB  = (tx*Bx+ty*By)/(fabs(T)*B);
  }
  d = material->GetDensity();
  cout<<"density "<<d<<"   name "<<material->GetName()<<endl;

  double Z      = material->GetZ();

  if (ZIncreasesWithTime == true) {
    dE =-BetheBloch * d * 1.e-1*fabs(T)*dz;
  } else {
    dE = BetheBloch * d * 1.e-1*fabs(T)*dz;
  }
  f  = (1.0 - 2.0*dE/(b*P));
  ag  = (1.0 - 1.0*dE/(b*P));

  double F40 = 0.0;
  double F41 = 0.0;
```

```
  double F42 = 0.0;
  double F43 = 0.0;
  double F44 = 0.0;

  double RangeZ = 0.0;
  double d1fdr1 = 0.0;
  double d2fdr2 = 0.0;
  double d3fdr3 = 0.0;
  double d4fdr4 = 0.0;
  double Nepsln = 0.0;

  if (Z == 6.0 || Z == 10.8046 || Z == 13.0 || Z == 26.0 || Z == 29.0) {
    RangeZ = IcalRange->MaterialMuRange(Z,P);
    d1fdr1 = IcalRange->FirstDerivative(RangeZ,Z);
    d2fdr2 = IcalRange->SecndDerivative(RangeZ,Z);
    d3fdr3 = IcalRange->ThirdDerivative(RangeZ,Z);
    d4fdr4 = IcalRange->FourthDerivative(RangeZ,Z);
  }
  if (Z == 6.0 || Z == 10.8046 || Z == 13.0 || Z == 26.0 || Z == 29.0) {
    //Nepsln = (d2fdr2/d1fdr1)*1.e2*d*dz;
    Nepsln = (d2fdr2/d1fdr1)*1.e2*d*T*dz + (d3fdr3/(2.0*d1fdr1))*pow(1.e2*d*T*dz
,2.0) + (d4fdr4/(6.0*d1fdr1))*pow(1.e2*d*T*dz,3.0);
  }

  F40 = d1fdr1 * kappa * qbyP *T* 1.e2 *d* dz * Bx;
  F41 = d1fdr1 * kappa * qbyP *T* 1.e2 *d* dz * By;
  F42 = d1fdr1 * (tx/T) * 1.e2 *d* dz;
  F43 = d1fdr1 * (ty/T) * 1.e2 *d* dz;

  if (Z == 6.0 || Z == 10.8046 || Z == 13.0 || Z == 26.0 || Z == 29.0) {
    F44 = 1.0 + Nepsln;
  } else {
    F44 = 1.0;
  }

  double Rx= Bx * dz * h;
  double Ry= By * dz * h;
  double Sx= 0.5 * Bx * pow(dz,2) * h;
  double Sy= 0.5 * By * pow(dz,2) * h;

  double tx_ty = tx*ty;
  double _1_3txsq_tysq = 1+3*pow(tx,2)+pow(ty,2);
  double _1_txsq_3tysq = 1+pow(tx,2)+3*pow(ty,2);
  double _1_3tysq = 1+3*pow(ty,2);
  double _1_3txsq = 1+3*pow(tx,2);
  double _1_txsq = 1+pow(tx,2);
  double _1_tysq = 1+pow(ty,2);
  double _4_6txsq_3tysq = 4+6*pow(tx,2)+3*pow(ty,2);
  double _4_3txsq_6tysq = 4+3*pow(tx,2)+6*pow(ty,2);
  double _1_6txsq_5tx4_tysq_3tx_tysq = 1+6*pow(tx,2)+5*pow(tx,4)+pow(ty,2)*3*pow
((tx_ty),2);
  double _1_6tysq_5ty4_txsq_3tx_tysq = 1+6*pow(ty,2)+5*pow(ty,4)+pow(tx,2)+3*pow
((tx_ty),2);
  double S_xx =(1/6)*Bx*Bx*pow(dz,3);
  double S_xy =(1/6)*Bx*By*pow(dz,3);
  double S_yx =(1/6)*By*Bx*pow(dz,3);
  double S_yy =(1/6)*By*By*pow(dz,3);
  double R_xx =(1/2)*Bx*Bx*pow(dz,2);
  double R_xy =(1/2)*Bx*By*pow(dz,2);
  double R_yx =(1/2)*By*Bx*pow(dz,2);
  double R_yy =(1/2)*By*By*pow(dz,2);

  //-------------------------------------------------------------------
  //cout<<"material "<<material->GetName()<<"    Z "<<material->GetZ()<<"    dz
"<<dz<<"    T "<<T<<"    tx "<<tx<<"    ty "<<ty<<"    qp "<<qbyP<<endl;
  //--------------------Beginning of first row:-----------------------

  F_k_minus[0][0] = 1.0 + (h/2.0)*(T*dz*dz)*(tx*ty*dBxdx-(1.0+tx*tx)*dBydx);
```



```
  F_k_minus[0][1] = (h/2.0)*(T*dz*dz)*(tx*ty*dBxdy-(1+tx*tx)*dBydy);

  F_k_minus[0][2] = dz+Sx*(((ty*pow(tx,2))/T2)+ty)-Sy*(((tx*(pow(tx,2)+1))/T2)+2
*tx)+
    pow(h,2) * ( ((_1_3txsq_tysq * _1_3tysq)/T2) * S_xx - 2*tx_ty*(_4_6txsq_3tys
q/T2) * S_xy - 2*tx_ty*(_4_6txsq_3tysq/T2) * S_yx + 3*(_1_6txsq_5tx4_tysq_3tx_ty
sq/T2)* S_yy);

  F_k_minus[0][3] = Sx*(((tx*pow(ty,2))/T2)+tx)-Sy*((ty*(pow(tx,2)+1))/T2)+
    pow(h,2) * ( 2*tx_ty*(_4_3txsq_6tysq/T2)* S_xx - ((_1_3txsq*_1_txsq_3tysq)/T
2) * S_xy - ((_1_3txsq*_1_txsq_3tysq)/T2) * S_yx + 6*tx_ty*(_1_txsq/T2)* S_yy);

  /*F_k_minus[0][4] = f*kappa*T*((0.5*Bx*pow(dz,2))*tx*ty-(0.5*By*pow(dz,2))*(po
w(tx,2)+1)) + 2*f*ag*h*kappa*T*( tx * _1_3tysq * S_xx - ty* _1_3txsq * S_xy - ty
* _1_3txsq * S_yx + 3 * tx *_1_txsq * S_yy);*/
  F_k_minus[0][4] = kappa*T*((0.5*Bx*pow(dz,2))*tx*ty-(0.5*By*pow(dz,2))*(pow(tx
,2)+1))+
    2 * qbyP * pow(kappa,2) * T2 * ( tx * _1_3tysq * S_xx - ty* _1_3txsq * S_xy
- ty* _1_3txsq * S_yx + 3 * tx *_1_txsq * S_yy);

  //------------------------End of first row-----------------------
  //*********************************************************
  //-------------------Beginning of second row:-------------------
  F_k_minus[1][0] = (h/2.0)*(T*dz*dz)*((1.0+ty*ty)*dBxdx-tx*ty*dBydx);

  F_k_minus[1][1] = 1.0 + (h/2.0)*(T*dz*dz)*((1+ty*ty)*dBxdy-tx*ty*dBydy);

  F_k_minus[1][2] = Sx*((tx*(pow(ty,2)+1))/T2)-Sy*(((ty*pow(tx,2))/T2)+ty)+
    pow(h,2) * ( 6 *tx_ty*(_1_tysq/T2)* S_xx - ((_1_3txsq_tysq*_1_3tysq)/T2)* S_
xy - ((_1_3txsq_tysq*_1_3tysq)/T2)* S_yx + (_4_6txsq_3tysq/T2)* S_yy);

  F_k_minus[1][3] = dz+Sx*(((ty*(pow(ty,2)+1))/T2)+2*ty)-Sy*(((tx*pow(ty,2))/T2)
+tx)+
    pow(h,2) * ( 3 * (_1_6tysq_5ty4_txsq_3tx_tysq/T2) * S_xx - 2 * tx_ty * (_4_3
txsq_6tysq/T2) * S_xy - 2 * tx_ty * (_4_3txsq_6tysq/T2) * S_yx + ((_1_3txsq_tysq
*_1_txsq_3tysq)/T2) * S_yy);

  /*F_k_minus[1][4] = f*kappa*T*((0.5*Bx*pow(dz,2))*(pow(ty,2)+1)-(0.5*By*pow(dz
,2))*(tx*ty)) + 2*f*ag*h*kappa*T*( 3*ty*(_1_tysq/T2)* S_xx - tx * (_1_3tysq/T2)
* S_xy - tx * (_1_3tysq/T2) * S_yx + ty *(_1_3txsq/T2) * S_yy);*/
  F_k_minus[1][4] = kappa*T*((0.5*Bx*pow(dz,2))*(pow(ty,2)+1)-(0.5*By*pow(dz,2))
*(tx*ty))+
    2 * qbyP * pow(kappa,2) * T2 * ( 3*ty*(_1_tysq/T2)* S_xx - tx * (_1_3tysq/T2
) * S_xy - tx * (_1_3tysq/T2) * S_yx + ty *(_1_3txsq/T2) * S_yy);

  //----------------------End of second row-----------------------
  //*********************************************************
  //-------------------Beginning of third row:--------------------
  F_k_minus[2][0] = T*h*dz*(tx*ty*dBxdx-(1.0+tx*tx)*dBydx);

  F_k_minus[2][1] = T*h*dz*(tx*ty*dBxdy-(1.0+tx*tx)*dBydy);

  F_k_minus[2][2] = (1+Rx*(ty+((ty*pow(tx,2))/T2))-Ry*(2*tx+((tx*(pow(tx,2)+1))/
T2))+
                   pow(h,2) * ( ((_1_3txsq_tysq * _1_3tysq)/T2) * R_xx - 2*tx_
ty*(_4_6txsq_3tysq/T2) * R_xy - 2*tx_ty*(_4_6txsq_3tysq/T2) * R_yx + 3*(_1_6txsq
_5tx4_tysq_3tx_tysq/T2)* R_yy));

  F_k_minus[2][3] = Rx*(tx+((tx*pow(ty,2))/T2))-Ry*((ty*(pow(tx,2)+1))/T2)+
    pow(h,2) * ( 2*tx_ty*(_4_3txsq_6tysq/T2)* R_xx - ((_1_3txsq*_1_txsq_3tysq)/T
2) * R_xy - ((_1_3txsq*_1_txsq_3tysq)/T2) * R_yx + 6*tx_ty*(_1_txsq/T2)* R_yy);

  /*F_k_minus[2][4] = f*kappa*T*((Bx*dz)*tx*ty-(By*dz)*(pow(tx,2)+1)) + 2*f*ag*h
*kappa*T*( tx * _1_3tysq * R_xx - ty* _1_3txsq * R_xy - ty* _1_3txsq * R_yx + 3
* tx *_1_txsq * R_yy);*/
  F_k_minus[2][4] = kappa*T*((Bx*dz)*tx*ty-(By*dz)*(pow(tx,2)+1))+
    2 * qbyP * pow(kappa,2) * T2 * ( tx * _1_3tysq * R_xx - ty* _1_3txsq * R_xy
- ty* _1_3txsq * R_yx + 3 * tx *_1_txsq * R_yy);
```

```
  //-----------------------End of third row-----------------------
  //**************************************************************
  //------------------Beginning of fourth row:--------------------
  F_k_minus[3][0] = T*h*dz*((1.0+ty*ty)*dBxdx-tx*ty*dBydx);

  F_k_minus[3][1] = T*h*dz*((1.0+ty*ty)*dBxdy-tx*ty*dBydy);

  F_k_minus[3][2] = Rx*((tx*(pow(ty,2)+1))/T2)-Ry*(ty+((ty*pow(tx,2))/T2))+
    pow(h,2) * ( 6 *tx_ty*(_1_tysq/T2)* R_xx - ((_1_3txsq_tysq*_1_3tysq)/T2)* R_
xy - ((_1_3txsq_tysq*_1_3tysq)/T2)* R_yx + (_4_6txsq_3tysq/T2)* R_yy);

  F_k_minus[3][3] = (1+Rx*(2*ty+((ty*(pow(ty,2)+1))/T2))-Ry*(tx+((tx*pow(ty,2))/
T2))+
                    pow(h,2) * ( 3 * (_1_6tysq_5ty4_txsq_3tx_tysq/T2) * R_xx -
2 * tx_ty * (_4_3txsq_6tysq/T2) * R_xy - 2 * tx_ty * (_4_3txsq_6tysq/T2) * R_yx
+ ((_1_3txsq_tysq*_1_txsq_3tysq)/T2) * R_yy));

  /*F_k_minus[3][4] = f*kappa*T*((Bx*dz)*(pow(ty,2)+1)-(By*dz)*(tx*ty)) + 2*f*ag
*h*kappa*T*( 3*ty*(_1_tysq/T2)* R_xx - tx * (_1_3tysq/T2) * R_xy - tx * (_1_3tys
q/T2) * R_yx + ty *(_1_3txsq/T2) * R_yy);*/
  F_k_minus[3][4] = kappa*T*((Bx*dz)*(pow(ty,2)+1)-(By*dz)*(tx*ty))+
    2 * qbyP * pow(kappa,2) * T2 * ( 3*ty*(_1_tysq/T2)* R_xx - tx * (_1_3tysq/T2
) * R_xy - tx * (_1_3tysq/T2) * R_yx + ty *(_1_3txsq/T2) * R_yy);

  //----------------------End of fourth row-----------------------
  //**************************************************************
  //-------------------Beginning of fifth row:--------------------

  F_k_minus[4][0] = F40;
  F_k_minus[4][1] = F41;
  F_k_minus[4][2] = F42;
  F_k_minus[4][3] = F43;
  F_k_minus[4][4] = F44;
  for(int ij=0; ij<5; ij++) {
    for(int jk = 0; jk<5; jk++) {
      bool za; za=std::isnan(F_k_minus[ij][jk]);
      if(za==true) { cout<<"F["<<ij<<"]["<<jk<<"]"<<endl;
      }
    }
  }
}

bool InoTrackFitAlg::PredictedStateCov(double *StateVector, const int Plane, int
 &NewPlane, const bool GoForward, double* x__minus, int isHalf, double* dS, doub
le* Range) { //, double* dE) {
  cout<<"=============== Entered InoTrackFitAlg::PredictedStateCov ==============="<<endl;

  cout<<"isHalf "<<isHalf<<endl;
  cout<<"NewPlane "<<NewPlane<<endl;
  cout<<"Plane "<<Plane<<endl;
  cout<<"GoForward "<<GoForward<<endl;



  int           DiffPlane;                                              DiffPlan
e = (NewPlane>0 && (!isHalf)) ? abs(NewPlane - Plane) : 1;
  cout<<"DiffPlane "<<DiffPlane<<endl;
  double epsilon=1.e-6;                                  //1micron shift in extra
polation

  //  double GPos[3]={0.0};
  double UxPos[3]={0.0};
  double UyPos[3]={0.0};
  double UxxPos[3]={0.0};
  double UyyPos[3]={0.0};
  double MPos[3]={0.0};
  double DxPos[3]={0.0};
  double DyPos[3]={0.0};
  double DxxPos[3]={0.0};
```



```
  double DyyPos[3]={0.0};
  double dBxdx = 0.0;
  double dBxdy = 0.0;
  double dBydx = 0.0;
  double dBydy = 0.0;

  //Mpos denote the starting (x,y,z) of the particle right at the RPC (mm)
  MPos[0]=StateVector[0] * 1000;
  MPos[1]=StateVector[1] * 1000;
  MPos[2]=ZPosLayer[Plane]*1.e3;

  cout<<"................................................................................"<<endl;
  cout<<"X "<<1.e-3*MPos[0]<<" Y "<<1.e-3*MPos[1]<<" Z"<<1.e-3*MPos[2]<<"   NIter "<
<NIter<<endl;
  cout<<"Plane number "<<Plane<<" to "<<NewPlane<<" "<<ZPosLayer[Plane]<<" with input P "<
<1./max(1.e-6,StateVector[4])<<"   N "<<NIter<<endl;
  for(int ij = 0; ij<5; ij++){cout<<"StateVector "<<ij<<" "<<StateVector[ij]<<endl;i
f(StateVector[ij]==0.0)cout<<"something wrong"<<"   "<<NIter<<"   "<<GoForward<<"   "<
<Plane<<"   "<<NewPlane<<"   "<<MinPlane<<"   "<<MaxPlane<<endl;}
  Double_t posTrack[3]={0.0};
  Double_t dirTrack[3]={0.0};
   cout<<"ZIncreasesWithTime "<<ZIncreasesWithTime<<endl;
  int initsign=(ZIncreasesWithTime != GoForward) ? -1 : 1;
  cout<<"initsign "<<initsign<<endl;
  int   nstep   = 0;
  //Do not go to infinite loop
  int   nstepmx = 100000000;
  double dzFe = 0;
  double dznF = 0;
        cout<<"DiffPlane "<<DiffPlane<<endl;
  for (int ij = 1; ij <= DiffPlane; ij++) {
          cout<<"DiffPlane "<<DiffPlane<<endl;
    double i1state[5];double i2state[5]; double i3state[5]={0.0};
    // i1state is the vector that InoTrackFitAlg() has just crossed RPC layer
    for (int jk=0; jk<5; jk++) {
      i1state[jk]= i2state[jk] =0;
    }

    // Feed the elements of the state vector (initially from TFinder
    // and from the next layer, coming from KFAlg) into i1state

    if (ij == 1) {
      for (int jk=0; jk<5; jk++) {
        i1state[jk] = StateVector[jk];  // i1state (here) is the vector just abo
ut to enter glass layer

        cout<<"i1state "<<jk<<" "<<i1state[jk]<<endl;

      }                                                          // normal case:
filtered state
    } else if (ij > 1) {
      cout<<"...................................................DiffPlane "<<DiffPlane<<" & n = "<<ij<<endl;
      for (int jk=0; jk<5; jk++) {
        i1state[jk] = x__minus[jk];      // i1state (here) is the vector just abo
ut to enter glass layer
      }                                                               // RPC h
it missed in the previous step
      for(int jk = 0; jk<5; jk++) {
        for(int kl = 0; kl<5; kl++) {
          C_k_minus[jk][kl]=C_k_intermediate[jk][kl];
        }
      }
    }

    for(int jk = 0; jk<5; jk++) {
      i3state[jk]=i1state[jk];

      cout<<"i3state "<<jk<<" "<<i3state[jk]<<endl;
    }
```

```
    double dzgl = 0;

    double UBx = 0.0;   double DBx = 0.0;               double UUBx = 0.0;
double DDBx = 0.0;
    double UBy = 0.0;   double DBy = 0.0;               double UUBy = 0.0;
double DDBy = 0.0;

    cout<<"ZPosLayer[2]"<<ZPosLayer[2]<<endl;
    cout<<"ZPosLayer[1]"<<ZPosLayer[1]<<endl;

    double totdist= (isHalf) ? 0.5*(ZPosLayer[2]-ZPosLayer[1]) : (ZPosLayer[2]-Z
PosLayer[1]);
    double f = 2.0;
    while(abs(dzgl)<totdist) {
      if (fabs(dzgl)> LayerThickness - 0.005) {
        break;
      }
      cout<<"MaxPlane"<<MaxPlane<<endl;
      cout<<"MinPlane"<<MinPlane<<endl;
      if (nHit<15 && abs(abs(MaxPlane-MinPlane)-nHit)<5 && (MaxPlane<140 && MinP
lane>10))
        f = 4.0;

      cout<<"i1state[2] "<<i1state[2]<<endl;
      cout<<"fabs(i1state[2]) "<<fabs(i1state[2])<<endl;

      if (fabs(i1state[2])>7.5) {
        cout<<"check 1"<<endl;
        i1state[2] = 7.5*(i1state[2]/fabs(i1state[2]));
      }
      if (fabs(i1state[3])>7.5) {
        cout<<"check 2"<<endl;
        i1state[3] = 7.5*(i1state[3]/fabs(i1state[3]));
      }
      if (fabs(i1state[4])>f) {
        cout<<"check 3"<<endl;
        i1state[4] = f * (i1state[4]/fabs(i1state[4]));
      }
      //GMAA temporary
      nstep++;
      cout<<"pre dzgl "<<dzgl<<endl;

      double dz=0;
      double Bx=0;
      double By=0;
      double dx=0;
      double dy=0;
      double Z =0;
      double Eloss = 0;
      double snext = 0;
      double density=0;

      int signp = 0;
      if (ZIncreasesWithTime==true) {
        signp = (ZIncreasesWithTime != GoForward) ? -1 :+1;
      } else {
        signp = (ZIncreasesWithTime != GoForward) ? +1 :-1;
      }

      double dxdz = i1state[2];
      double dydz = i1state[3];
      double dsdz = pow((1.+pow(i1state[2],2)+pow(i1state[3],2)),0.5);

      dirTrack[0] = signp*(dxdz/dsdz);
      dirTrack[1] = signp*(dydz/dsdz);
      dirTrack[2] = signp/dsdz;

      for (int jk=0; jk<3; jk++) {
```



```
        posTrack[jk] = 0.1 * MPos[jk]+epsilon*dirTrack[jk];           //conver
t mm to cm
      }

      icalGeometry->InitTrack(posTrack, dirTrack);

      Bx = By = 0.0;
      pFieldMap->ElectroMagneticField(MPos,Bx,By,1);
      Bx *= 1000;
      By *= 1000;

      localmat= icalGeometry->GetCurrentVolume()->GetMaterial();
      icalGeometry->FindNextBoundary();
      snext     = epsilon+gGeoManager->GetStep();
      Z         = localmat->GetZ();
      cout<<" material "<<localmat->GetName()<<"   Z "<<Z<<"   A "<<localmat->GetA()<<
endl;
      if (snext > 1.0) snext=1.0;

      if (strstr(icalGeometry->GetCurrentVolume()->GetName(),"IRLAYElog")) {
        dz = signp*min(0.001, 0.01*snext*abs(dirTrack[2]));
        cout<<" material "<<localmat->GetName()<<"   Z "<<Z<<"   A "<<localmat->GetA()
<<endl;
        double d = 1.0/sqrt(1.0+i2state[2]*i2state[2]+i2state[3]*i2state[3]);

        UxPos[0] = MPos[0] + d;         DxPos[0] = MPos[0] - d;
UyPos[0] = MPos[0];                     DyPos[0] = MPos[0];
        UxPos[1] = MPos[1];                     DxPos[1] = MPos[1];
                UyPos[1] = MPos[1] + d;         DyPos[1] = MPos[1] - d;
        UxPos[2] = MPos[2];                     DxPos[2] = MPos[2];
                UyPos[2] = MPos[2];                     DyPos[2] = MPos[2];

        UxxPos[0] = MPos[0] + 2*d;      DxxPos[0] = MPos[0] - 2*d;
UyyPos[0] = MPos[0];            DyyPos[0] = MPos[0];
        UxxPos[1] = MPos[1];            DxxPos[1] = MPos[1];
UyyPos[1] = MPos[1] + 2*d;      DyyPos[1] = MPos[1] - 2*d;
        UxxPos[2] = MPos[2];            DxxPos[2] = MPos[2];
UyyPos[2] = MPos[2];            DyyPos[2] = MPos[2];

        //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        pFieldMap->ElectroMagneticField(UxPos,UBx,UBy,1);
        pFieldMap->ElectroMagneticField(DxPos,DBx,DBy,1);

        pFieldMap->ElectroMagneticField(UxxPos,UUBx,UUBy,1);
        pFieldMap->ElectroMagneticField(DxxPos,DDBx,DDBy,1);

        //O(h^2) order derivatives of B field
        //dBxdx = 1.e3*(UBx-DBx)/(2*d*1.e-3);
        //dBydx = 1.e3*(UBy-DBy)/(2*d*1.e-3);

        //O(h^4) order derivatives of B field
        dBxdx = (1.e3*(-UUBx + 8.0*UBx - 8.0*DBx + DDBx))/(1.e-3*(12.0*d));
        dBydx = (1.e3*(-UUBy + 8.0*UBy - 8.0*DBy + DDBy))/(1.e-3*(12.0*d));
        cout<<"xxis this iron? "<<Z<<endl;
        pFieldMap->ElectroMagneticField(UyPos,UBx,UBy,1);
        pFieldMap->ElectroMagneticField(DyPos,DBx,DBy,1);

        pFieldMap->ElectroMagneticField(UyyPos,UUBx,UUBy,1);
        pFieldMap->ElectroMagneticField(DyyPos,DDBx,DDBy,1);

        //O(h^2) order derivatives of B field
        //dBxdy = 1.e3*(UBx-DBx)/(2*d*1.e-3);
        //dBydy = 1.e3*(UBy-DBy)/(2*d*1.e-3);

        //O(h^4) order derivatives of B field
        dBxdy = (1.e3*(-UUBx + 8.0*UBx - 8.0*DBx + DDBx))/(1.e-3*(12.0*d));
        dBydy = (1.e3*(-UUBy + 8.0*UBy - 8.0*DBy + DDBy))/(1.e-3*(12.0*d));
        cout<<"is this iron? "<<Z<<endl;
      } else {
```

```
        cout <<"MPos "<< MPos[0]<<"  "<< MPos[1]<<"  "<< MPos[2]<<"  "<<Bx<<"  "
<<By<<"  "<<icalGeometry->GetCurrentVolume()->GetName()<<endl;
        Bx = By = 0.0;

        dBxdx = 0.0;    dBxdy = 0.0;
        dBydx = 0.0;    dBydy = 0.0;

        //if (localmat->GetName()=="G4_Air")
        //dz = 0.01*signp;
        //else
        cout<<"is this !iron? "<<Z<<endl;
        dz = 0.01*signp*snext*abs(dirTrack[2]); // cm to meter
        if(strstr(icalGeometry->GetCurrentVolume()->GetName(),"GASRlog")) {
          dz = 0.001*signp;
        }
        cout<<"is this !iron? "<<Z<<endl;
      }

      //if (nstep>=nstepmx) {dz= totdist - abs(dzgl);cout<<"00000000000000000000
0000000000"<<endl;}
      dzgl +=dz;

      //DZ[nstep] = dz;
      if (Z == 26.0) {
        dzFe += dz;
      }
      if ((NIter == 1) && (GoForward==true) && (NewPlane-MinPlane)==1) {
        B_in = sqrt(Bx*Bx + By*By);
      }
      if (Z != 26.0) {
        dznF += dz;
      }
      density = localmat->GetDensity();
      cout<<"dz "<<dz<<" dzgl "<<dzgl<<" snext "<<snext<<" material "<<icalGeometry->GetC
urrentVolume()->GetName()<<" density"<<density<<endl;

      Eloss = GetEnergyLoss(i1state, dz, xi, T_max, I, localmat);
      cout <<"Elose "<<Eloss<<endl;
      //if (localmat->GetName()=="G4_Air")
      //if(strstr(icalGeometry->GetCurrentVolume()->GetName(),"LAYElog"))
      //Eloss = 0.0;

      /*
      // Find the propagator
      GetPropagator(i1state, Bx, By, dBxdx, dBydx, dBxdy, dBydy, dz, localmat);

      // Find the multiple scattering in the step, with given slope
      bool uv;
      for (int jk = 0; jk< 5; jk++) {
      uv = std::isnan(i1state[jk]);
      }

      if (NewPlane>=0 && uv==false)
      GetMultipleScattering(i1state, Bx, By, dz, xi, T_max, I, localmat);
      */
      //GMAA Use Fmatrix to propagate
      // Get the state elements
      double x; double y; double tx; double ty; double qbyP;
      x =i1state[0];
      y =i1state[1];
      tx        =i1state[2];
      ty        =i1state[3];
      qbyP=i1state[4];

      // Construction of variables needed for 3rd order extrapolation
      double kappa      = 0.299792458;
      double T          = sqrt(1+pow(tx,2)+pow(ty,2));          if (ZIncreasesWi
thTime==false) T=-T;
      double h          = kappa*qbyP*T;
```



```
      double Rx         = Bx*dz*h;
      double Ry         = By*dz*h;
      double Sx         = 0.5*Bx*pow(dz,2)*h;
      double Sy         = 0.5*By*pow(dz,2)*h;
      double Rxx = 0.5 * Bx * Bx * pow(dz,2);   double Sxx = (1/6)*Bx*Bx*pow(dz,
3);
      double Rxy = 0.5 * Bx * By * pow(dz,2);   double Sxy = (1/6)*Bx*By*pow(dz,
3);
      double Ryx = 0.5 * By * Bx * pow(dz,2);   double Syx = (1/6)*By*Bx*pow(dz,
3);
      double Ryy = 0.5 * By * By * pow(dz,2);   double Syy = (1/6)*By*By*pow(dz,
3);

      double Bi = sqrt(Bx*Bx + By*By + pow((tx*By-ty*Bx)/T,2.0));

      double P = 0;     double P_ex = 0;        double E = 0;   double E_ex = 0;
double mumas = 0.1056589;       double b = 0.0; double ddE = 0.0;

      if (qbyP == 0) {
        i2state[4] = qbyP;
      } else if (qbyP != 0) {
        P       = fabs(1.0/qbyP);       //cout<<"PSC(iron) P = "<<P<<endl;
        E       = sqrt(pow(P,2) + pow(mumas,2));   //cout<<"PSC(iron) E = "<<E<<
endl;
        b       = P/E;

        if (ZIncreasesWithTime == true) {
          ddE =-BetheBloch * density * 1.e-1*fabs(T)*dz;
        } else {
          ddE = BetheBloch * density * 1.e-1*fabs(T)*dz;
        }

        if (ZIncreasesWithTime==true && dz > 0) {
          E_ex= E - Eloss;      //cout<<"PSC(going up in iron) E_ex = "<<E_ex<<e
ndl;
        } else if (ZIncreasesWithTime==true && dz < 0) {
          E_ex= E + Eloss;      //cout<<"PSC(going down in iron) E_ex = "<<E_ex<
<endl;
        } else if (ZIncreasesWithTime==false && dz < 0) {
          E_ex= E - Eloss;      //cout<<"a"<<endl;
        } else if (ZIncreasesWithTime==false && dz > 0) {
          E_ex= E + Eloss;      //cout<<"b"<<endl;
        }

        if ((E_ex-mumas)>0) {
          P_ex= sqrt(E_ex*E_ex - mumas*mumas);  //cout<<"PSC(iron) P_ex = "<<P_e
x<<endl;
        }
        i2state[4] = qbyP * (P/P_ex);

        //i2state[4] = qbyP - qbyP*(ddE/(b*P));
      }
      cout<<"P "<<P<<"   P_ex "<<P_ex<<"   dz "<<dz<<"   Eloss "<<Eloss<<endl;

      i2state[0] = x + tx * dz + tx * ty * Sx - (pow(tx,2)+1) * Sy + h*h * (tx*(
3*ty*ty+1)*Sxx - ty*(3*tx*tx+1)*Sxy -ty*(3*tx*tx+1)*Syx + tx*(3*tx*tx+3)*Syy);
      i2state[1] = y + ty * dz + (pow(ty,2)+1) * Sx - tx * ty * Sy + h*h * (ty*(
3*ty*ty+3)*Sxx - tx*(3*ty*ty+1)*Sxy -tx*(3*ty*ty+1)*Syx + ty*(3*tx*tx+1)*Syy);
      i2state[2] = tx + tx * ty * Rx - (pow(tx,2)+1) * Ry + h*h * (tx*(3*ty*ty+1
)*Rxx - ty*(3*tx*tx+1)*Rxy -ty*(3*tx*tx+1)*Ryx + tx*(3*tx*tx+3)*Ryy);
      i2state[3] = ty + (pow(ty,2)+1) * Rx - tx * ty * Ry + h*h * (ty*(3*ty*ty+3
)*Rxx - tx*(3*ty*ty+1)*Rxy -tx*(3*ty*ty+1)*Ryx + ty*(3*tx*tx+1)*Ryy);

      bool bull; bull = std::isnan(P_ex);
      if (bull == true) {
        cout<<"Not-A-Number occurring. Event must be stopped"<<endl;
        break;
      }
```

```
      dx = (i2state[0] - i1state[0]);
      dy = (i2state[1] - i1state[1]);

      double length = sqrt(dx*dx + dy*dy + dz*dz);

      cout <<"ds "<<*dS<<endl;
      *dS += length;
      *Range += BetheBloch * density * (1.e2*length) * 1.e-3;
      cout <<"Range "<<*Range<<endl;


      double Tx = i2state[2];
      double Ty = i2state[3];
      double TT = sqrt(1.0 + Tx*Tx + Ty*Ty);

      double Ps[3]={0.0000000};
      Ps[0]     = 1.e3*i2state[0];
      Ps[1]     = 1.e3*i2state[1];
      Ps[2]     = 0.0000000000000;
      double bX = 0.000000000;
      double bY = 0.000000000;
      pFieldMap->ElectroMagneticField(Ps,bX,bY,1);
      bX = 1.e3*bX;bY = 1.e3*bY;
      double Bf = sqrt(bX*bX + bY*bY + pow((Tx*bY-Ty*bX)/TT,2.0));
      MagicRatio =0.000;
      if (Bx!= 0.0 && By != 0.0) {
        MagicRatio = Bi/Bf;
      }
      //.........................................................................
........................................//
      double dbxdx = 0.000000;          double dbydx = 0.0000000000;
double dbxdy = 0.0000000000;    double dbydy=0.;
      //.........................................................................
........................................//
      /*          x + dx         */              /*          y            */
      double exPos_x[3]={0.0};          exPos_x[0] = 1.e3*i1state[0]+(tx/fabs(tx
))*50.0;              exPos_x[1] = 1.e3*i1state[1];   exPos_x[2] = 0.0;
//exPos_x[2] = MPos[2]+1.e3*dz;
      //.........................................................................
........................................//
      /*             x           */              /*        y + dy        */
      double exPos_y[3]={0.0};          exPos_y[0] = 1.e3*i1state[0];
exPos_y[1] = 1.e3*i1state[1]+(ty/fabs(ty))*50.0;        exPos_y[2] = 0.0;
//exPos_y[2] = MPos[2]+1.e3*dz;
      //.........................................................................
........................................//
      double bx = 0.0;
      double by = 0.0;

      if (strstr(icalGeometry->GetCurrentVolume()->GetName(),"IRLAYElog")) {//lo
calmat->GetName()=="G4_Fe"
        pFieldMap->ElectroMagneticField(exPos_x,bx,by,1);
        bx = 1.e3*bx;
        by = 1.e3*by;
        dbxdx = (bx-Bx)/((tx/fabs(tx))*0.05);
        dbydx = (by-By)/((tx/fabs(tx))*0.05);
        //cout<<"X: dbxdx "<<dbxdx<<"     dbxdy "<<dbxdy<<endl;
        bx = 0.0;
        by = 0.0;

        pFieldMap->ElectroMagneticField(exPos_y,bx,by,1);
        bx = 1.e3*bx;
        by = 1.e3*by;
        dbxdy = (bx-Bx)/((ty/fabs(ty))*0.05);
        dbydy = (by-By)/((ty/fabs(ty))*0.05);
        //cout<<"Y: dbydx "<<dbydx<<"     dbydy "<<dbydy<<endl;
      }

      cout<<"dBx "<<dbxdx*dx + dbxdy*dy<<"  Bx "<<Bx<<"  dBy "<<dbxdx*dx + dbxdy*d
```



```
y<<"  By "<<By<<endl;
      cout<<"........................"<<endl;

      // Find the propagator
      GetPropagator(i1state, Bx, By, dBxdx, dBydx, dBxdy, dBydy, dz, localmat);
      cout<<"material "<<localmat->GetName()<<"  Z "<<localmat->GetZ()<<"  dz "<<dz<
<" inside PSC"<<endl;
      /*
        cout<<"F00 "<<F_k_minus[0][0]<<"     F01 "<<F_k_minus[0][1]<<"     F02 "
<<F_k_minus[0][2]<<"     F03 "<<F_k_minus[0][3]<<"     F04 "<<F_k_minus[0][4]<<"
\n"
        <<"F10 "<<F_k_minus[1][0]<<"     F11 "<<F_k_minus[1][1]<<"     F12 "<<F_
k_minus[1][2]<<"     F13 "<<F_k_minus[1][3]<<"     F04 "<<F_k_minus[1][4]<<"\n"
        <<"F20 "<<F_k_minus[2][0]<<"     F21 "<<F_k_minus[2][1]<<"     F22 "<<F_
k_minus[2][2]<<"     F23 "<<F_k_minus[2][3]<<"     F04 "<<F_k_minus[2][4]<<"\n"
        <<"F30 "<<F_k_minus[3][0]<<"     F31 "<<F_k_minus[3][1]<<"     F32 "<<F_
k_minus[3][2]<<"     F33 "<<F_k_minus[3][3]<<"     F04 "<<F_k_minus[3][4]<<"\n"
        <<"F40 "<<F_k_minus[4][0]<<"     F41 "<<F_k_minus[4][1]<<"     F42 "<<F_
k_minus[4][2]<<"     F43 "<<F_k_minus[4][3]<<"     F04 "<<F_k_minus[4][4]<<"\n"
        <<endl;
        cout<<".........................."<<endl;
      */
      // Find the multiple scattering in the step, with given slope
      bool uv;
      for (int jk = 0; jk< 5; jk++) {
        uv = std::isnan(i1state[jk]);
      }

      if (NewPlane>=0 && uv==false) {
        GetMultipleScattering(i1state, Bx, By, dz,/* xi,*/ T_max, I, localmat);
      }
      // Extrapolation of the covariance matrix: [C^(k-1)]_k = F_k * C_(k-1) * (
F_k)^T + Q_(k-1)
      if (NewPlane >=0)
        ExtrapCovMatrix();

      for (int jk = 0; jk < 5; jk++) {
        for (int kl = 0; kl < 5; kl++) {
          C_k_minus[jk][kl]     = C_k_intermediate[jk][kl];
          if (Z == 6.0 || Z == 10.8046 || Z == 13.0 || Z == 26.0 || Z == 29.0) {
            if (!fMT) {
              C_k_minus[jk][kl]+= 1.0*Q_k_minus[jk][kl];
            } else {
              C_k_minus[jk][kl]+= 1.0*Q_k_minus[jk][kl];
            }
          }
        }
      }

      // Set the (x,y,z) of the extrapolated track, for calling the local magnet
ic field
      MPos[0]=i2state[0]*1000;                          //MPos[0]=GPos[0];
      MPos[1]=i2state[1]*1000;                          //MPos[1]=GPos[1];
      MPos[2]=1.e3*(1.e-3*MPos[2]+dz);          //MPos[2]=GPos[2]*1000;

      for (int jk=0; jk<5; jk++) {
        i1state[jk] = 0.0;
        i1state[jk] = i2state[jk];
      }

      //if (fabs(dz)>0.02)
      //cout<<"dz "<<dz<<" Z pos "<<MPos[2]<<"     snext "<<snext<<"     nstep "
<<nstep<<"     material "<<localmat->GetName()<<"     previous step size dz "<<D
Z[nstep-1]<<"     Plane "<<Plane<<"     NewPlane "<<NewPlane<<endl;

      //cout<<"post dzgl "<<dzgl<<endl;
    }
    // while loop (controlling dZ <= 0.096 m) ends here
```

```
    if (NewPlane>=0) {
      //cout<<"..........................................."<<endl;
      for (int jk=0; jk<5; jk++) {
        x__minus[jk] = i1state[jk];
      }
    }
    for (int jk = 0; jk < 5; jk++)      {
      for (int kl = 0; kl < 5; kl++) {
        C_k_intermediate[jk][kl] = 0;
        C_k_intermediate[jk][kl] = C_k_minus[jk][kl];
      }
    }
  }// for loop ends here
  //cout<<dzFe<<"     "<<dznF<<endl;
  cout<<"NewPlane "<<NewPlane<<"ZIncreasesWithTime "<<ZIncreasesWithTime<<"nstep "<<nste
p<<endl;
  if (NewPlane<0 && nstep<nstepmx){
    if (ZIncreasesWithTime==true) {
      NewPlane = Plane + initsign;
    } else {
      NewPlane = Plane - initsign;
      cout<<"NewPlane_ "<<NewPlane<<"ZIncreasesWithTime "<<ZIncreasesWithTime<<"nstep "<
<nstep<<endl;
    }
  }
  cout<<"yyy "<<GPL<<" "<<RNG<<" "<<endl;
  /*
   *dS = GPL;   //GMA14 Put all these three properly
   *Range = RNG;
   *dE = 0;
   cout<<"xxx "<<endl;
  */

  return true;
}

void InoTrackFitAlg::ExtrapCovMatrix() {
  // C_k_intermediate = (F_k_minus * C_k_minus * F_k_minus^T) + Q_k_minus
  if(debug_fit) {cout <<" ----------------------InoTrackFitAlg : ExtrapCovMatrix-----------
-----------" << endl;}

  for (int ij=0; ij<5; ++ij) {
    for (int jk=0; jk<5; ++jk) {
      C_k_intermediate[ij][jk]=0;
      for (int kl=0; kl<5; ++kl) {
        for (int lm=0; lm<5; ++lm) {
          C_k_intermediate[ij][jk]+=F_k_minus[ij][lm]*C_k_minus[lm][kl]*F_k_minu
s[jk][kl];
        }
      }
    }
  }

  //for(int ij=0; ij<5; ij++){for(int jk = 0; jk<5; jk++) {bool z; z=std::isnan(
F_k_minus[ij][jk]); if(z==true) cout<<"F"<<endl;}}
  //for(int ij=0; ij<5; ij++){for(int jk = 0; jk<5; jk++) {bool z; z=std::isnan(
C_k_minus[ij][jk]); if(z==true) cout<<"C"<<endl;}}
  //for(int ij=0; ij<5; ij++){for(int jk = 0; jk<5; jk++) {bool z; z=std::isnan(
C_k_intermediate[ij][jk]); if(z==true) cout<<"FCF^"<<endl;}}
  /*
  // Diagonal elements should be positive
  double covlim = 1.e-8;

  for(int ij=0; ij<5; ++ij) {
  if(C_k_intermediate[ij][ij]<covlim){
  // GMA reopen this      cout <<" InoTrackFitAlg : Negative diagonal element in
 C_k_intermediate" << endl;
  // C_k_intermediate[ij][ij]=covlim;
  }
```



```
  }

  // Display
  if(debug_fit) {
    cout<<"-------------------------------------------------------------"<<endl;
    cout << "C_k_intermediate" << endl;
    for(int ij=0; ij<5; ++ij) {
      for(int jk=0; jk<5; ++jk) {
        cout << C_k_intermediate[ij][jk] << " ";
      }
      cout << endl;
    }
    cout<<"-------------------------------------------------------------"<<endl;
  }
  for (int ij=0; ij<5; ij++) {
    for (int jk =0; jk<5; jk++)  {
      C_k_minus[ij][jk] = C_k_intermediate[ij][jk];
    }
  }
  */
}

void InoTrackFitAlg::ExtrapCovMatrixall() {
  // C_k_intermediate = (F_k_minus * C_k_minus * F_k_minus^T) + Q_k_minus
  if(debug_fit) {cout <<" ----------------------InoTrackFitAlg : ExtrapCovMatrix-----------
-----------" << endl;}

  for (int ij=0; ij<5; ++ij) {
    for (int jk=0; jk<5; ++jk) {
      C_k_intermediate[ij][jk]=0;
      for (int kl=0; kl<5; ++kl) {
        for (int lm=0; lm<5; ++lm) {
          C_k_intermediate[ij][jk]+=F_k_minus[ij][lm]*C_k_minus[lm][kl]*F_k_minu
s[jk][kl];
        }
      }
    }
  }

  for (int ij=0; ij<5; ij++) {
    for (int jk =0; jk<5; jk++)  {
      C_k_minus[ij][jk] = C_k_intermediate[ij][jk];
    }
  }
}

/*
  void InoTrackFitAlg::CovarianceMatrixExtrapolation() {
  for (int ij=0; ij<5; ++ij) {
  for (int jk=0; jk<5; ++jk) {
  C_k_intermediate[i][jk]=0;
  for (int kl=0; kl<5; ++kl) {
  for (int lm=0; lm<5; ++lm) {
  // intermediate C_k is calculated at each step from intermediate F_k_minus
  C_k_intermediate[ij][jk]+=F_k_minus[ij][lm]*C_k_minus[lm][kl]*F_k_minus[jk][kl
];
  }
  }
  }
  }
  }
*/
double InoTrackFitAlg::GetEnergyLoss(double *istate, double dz, double &axi, dou
ble &aT_max, double &aI, TGeoMaterial* material) {
  // cout<<"*****Energy Loss Function*****"<<endl;
  // Initialize the relevant elements of the state vector so that fresh values m
ay fed to appropriate variables.
  double tx = 0; double ty = 0; double qbyP = 0; double P = 0;
```

```
  tx=istate[2];
  ty=istate[3];
  qbyP=istate[4];

  double T =sqrt(1+pow(tx,2)+pow(ty,2));

  ///////////////////////////////////////////////////////////////////////////
  // The following part will produce the Bethe Bloch energy loss at momentum P
  ///////////////////////////////////////////////////////////////////////////
  // Setting Momentum and Energy of the muon currently

  if (fabs(qbyP) < 0.01) {
    P = 100.0;  // GeV
  } else if (fabs(qbyP) >= 0.01 && fabs(qbyP) <= 4.0) {
    P = fabs(1/qbyP);  // GeV
  } else {
    P = 0.25;
  }
  // Here P is found in GeV/c

  double em     = 0.5110;               // MeV/c^2
  double uam    = 105.65839;    // Mev/c^2

  double a      = 0.0;
  double am     = 0.0;
  double X0     = 0.0;
  double X1     = 0.0;
  double Cbar= 0.0;
  double dltO= 0.0;
  aI                    = 0.0;

  std::map<string, double> meanExEnergy;
  meanExEnergy["G4_Fe"]                = 286.0E-6;
  meanExEnergy["G4_Cu"]                = 322.0E-6;
  meanExEnergy["G4_Al"]                = 166.0E-6;
  meanExEnergy["G4_AIR"]               = 85.7E-6;
  meanExEnergy["G4_POLYETHYLENE"]     = 57.4E-6;
  meanExEnergy["G4_MYLAR"]            = 78.7E-6;
  meanExEnergy["G4_SILICON_DIOXIDE"] = 139.2E-6;
  meanExEnergy["G10"]                 = 72.4E-6; //same as bakalite
  meanExEnergy["G4_GRAPHITE"]         = 78.0E-6;
  meanExEnergy["rpcgas"]             = 85.7E-6; //same as air
  meanExEnergy["FRPCarbon"]         = 78.0E-6; // same as graphite
  aI = meanExEnergy[material->GetName()];

  double Z      = material->GetZ();
  double A      = material->GetA();
  double rho    = material->GetDensity();                              // g/cm^
3

  // Sternheimer's coefficients:
  if (Z==6 && rho ==2.21) {
    a = 0.2614;         am = 2.8697;            X0 =-0.0178;            X1 = 2.3
415;            Cbar = 2.8680;          dltO = 0.12;
  } else if (Z==13) {
    a = 0.0802;         am = 3.6345;            X0 = 0.1708;            X1 = 3.0
127;            Cbar = 4.2395;          dltO = 0.12;
  } else if (Z==26) {
    a = 0.1468;         am = 2.9600;            X0 =-0.0012;            X1 = 3.1
531;            Cbar = 4.2911;          dltO = 0.12;
  } else if (Z==29) {
    a = 0.1434;         am = 2.9044;            X0 =-0.0254;            X1 = 3.2
792;            Cbar = 4.4190;          dltO = 0.08;
  }

  // Convert Momentum to MeV/c
  P = 1.e3 * P;

  double E = 0;     // MeV
```



```
  E = sqrt(pow(P,2) + pow(uam,2)); // MeV

  double beta = 0;
  beta = P/E;

  double gamma = 0;
  gamma = E/uam;

  double eta = 0;
  eta = beta * gamma;

  double X = 0;
  X = log10 (eta);

  double delta = 0;
  // Density Effect correction (Using Sternheimer's parametrization)

  if (Z==6.0 || Z == 26.0 || Z == 29.0 || Z == 13.0) {
    if (log10(eta) >= X1) {
      delta     = 2 * log(10) * log10(eta) - Cbar;
    } else if (log10(eta) > X0 && log10(eta) < X1) {
      delta     = 2 * log(10) * log10(eta) - Cbar + a * pow((X1-log10(eta)),am);
    } else {
      delta     = dltO * pow(10.0,2*(X-X0));
    }
    if (Z==6 && log10(eta) < X0) {
      delta = 0.0;
    }
  } else {
    delta = 0.0;
  }

  //double BetheBloch;
  //////////////////////////////////////////////////////////////////////
  // Bethe-Bloch: ionization loss
  BetheBloch = 0.15353 * (Z/A) * (1/pow(beta,2)) * (log((4 * pow(em,2) * pow(eta
,4))/(pow(aI,2)*(1 + (em/uam)*sqrt(1+pow(eta,2)))))-2*pow(beta,2)-delta);
  // found in MeV. cm^2/g
  //cout<<"Material "<<Z<<"P "<<P<<"     "<<"BetheBloch = "<<BetheBloch<<endl;
  //////////////////////////////////////////////////////////////////////

  double dl     = 0; // cm
  double rds    = 0; // g/cm^2

  dz = 100 * dz;                // z-step; dz expressed in cm
  dl = fabs(T*dz);      // total length; found in cm
  rds = rho * dl;               // rds is found in the units of g/cm^2
  //cout<<"rds (g/cm^2) = "<<rds<<endl;

  // [MeV]     BetheBloch        rds
  // Eloss = (MeV*cm^2 / g) * (g/cm^2) * [T] = MeV = GeV/1000

  double Eloss;
  Eloss = 0.0;
  axi           = 0;
  aT_max        = 0;

  // Eventually we will treat tracks going through a support structure
  // in a function different from PredictedStateCov. Here we deal with
  // those tracklets for which (NewPlane - Plane) = 1. For these track
  // -lets, the following conition is fine to find out the mean energy
  // loss:

  Eloss = BetheBloch * rds; // this is found in MeV
  Eloss = 1.e-3 * Eloss; // GeV
  axi           = 0.15353*(Z/A) * (rds/pow(beta,2));    // MeV
  axi           = 1.e-3 * axi;
  aT_max        = (2 * em * pow(eta,2))/(1 + 2 * (em/uam) * sqrt(1 + pow(eta,2))
);
```

```
  aT_max        = 1.e-3 * aT_max;

  return Eloss;
}

void InoTrackFitAlg::GetMultipleScattering(double* i2state,double Bx, double By,
 double dz,/* double xi,*/ double aT_max, double aI, TGeoMaterial* material) {
  //cout<<"*****MS matrix*****"<<endl;

  double tx = 0; double ty = 0; double qbyP = 0; // general elements of the stat
e vector

  tx=i2state[2];
  ty=i2state[3];
  qbyP=i2state[4];

  double T=sqrt(1+pow(tx,2)+pow(ty,2));
  double kappa  = 0.299792458;
  double uam = 0.1056;   // GeV
  double P = 0;
  double E = 0;
  double Z = material->GetZ();
  double dn= material->GetDensity();

  if (fabs(qbyP) < 0.01) {
    P = 100.0;  // GeV
  } else if (fabs(qbyP) >= 0.01 && fabs(qbyP) <= 4.0) {
    P = fabs(1/qbyP);   // GeV
  } else {
    P = 0.25;
  }
  E = sqrt(pow(P,2) + pow(uam,2)); // GeV

  double beta = 0;
  beta = P/E;

  double gamma = 0;
  gamma = E/uam;

  double eta = 0;
  eta = beta * gamma;

  double h = kappa * qbyP * T;

  double Rx= Bx * dz * h;
  double Ry= By * dz * h;
  double Sx= 0.5 * Bx * pow(dz,2) * h;
  double Sy= 0.5 * By * pow(dz,2) * h;

  //  double Lkappa = xi/aT_max;
  double Q55 = 0;

  // Q55 = [ÏM-^C(P)]^2/P^4 = (E^2/P^6)*[ÏM-^C(E)]^2

  // if (Lkappa <= 0.05)
  // Q55 = ((pow(E,2))/(pow(P,6))) * pow((100.0 * xi),2);
// Landau
  // else
  // Q55 = ((pow(E,2))/(pow(P,6))) * (1 - pow(beta,2)/2) * xi * aT_max;// Gaussi
an

  //Q55 = (0.25 * (xi/(P*P)) * T)*(0.25 * (xi/(P*P)) * T);
// MINOS (John Marshall)

  double f2             = 2./Z;
  double f1             = 1-f2;
  double e2             = 10.0*(Z*Z);
//eV
  double e1             = pow(((aI*1.e6)/pow(e2,f2)),(1./f1));  //(eV?) mean exc
```



```
itation energy "aI" converted to eV, then formula applied
  double r              = 0.4;
  double dEds   = BetheBloch * dn;
//MeV/cm
  aI                            = 1.e-3 *aI;
              //GeV

  double SIGMA_1        = dEds * (1-r) * (f1*(log(2*uam*eta*eta/(1.e-9*e1))-beta
*beta))/(1.e-6*e1*(log(2*uam*eta*eta/aI)-beta*beta));
  double SIGMA_2        = dEds * (1-r) * (f2*(log(2*uam*eta*eta/(1.e-9*e2))-beta
*beta))/(1.e-6*e2*(log(2*uam*eta*eta/aI)-beta*beta));
  double SIGMA_3        = dEds * r * (1.0/(1.e3*aI*(1+aI/aT_max)))*(1.0/log(1+aT
_max/aI));

  double alpha  = 0.0;
  alpha = 0.997;

  double Ealpha = (aI*(aI + aT_max))/(aI + (1-alpha)*aT_max);
  double E_bar  = (1 + aI/aT_max) * aI * log(Ealpha/aI);
//((aI*(aI + aT_max))/aT_max)*log(Ealpha/aI);
  double E_sq_bar= (1 + aI/aT_max) * aI * (Ealpha - aI);
  double SgmAlpSq= E_sq_bar - E_bar*E_bar;

  int n1 = PoissonRn->Poisson(SIGMA_1);
  int n2 = PoissonRn->Poisson(SIGMA_2);
  int n3 = PoissonRn->Poisson(SIGMA_3);

  e1                            = 1.e-9*e1;
  e2                            = 1.e-9*e2;
  double SigmaSqE= n1 * e1*e1 + n2 * e2*e2 + n3 * E_bar*E_bar + n3 * SgmAlpSq *
(n3 + 1);

  Q55 = ((pow(E,2))/(pow(P,6))) * SigmaSqE;
  //Q55 = ((pow(E,2))/(pow(P,6))) * pow((75.0 * xi),2);

  /*    if (Z==26.0){cout<<"SigmaSqE "<<SigmaSqE<<endl;
        cout<<"n1 "<<n1<<"     SIGMA1 "<<SIGMA_1<<endl;
        cout<<"n2 "<<n2<<"     SIGMA2 "<<SIGMA_2<<endl;
        cout<<"n3 "<<n3<<"     SIGMA3 "<<SIGMA_3<<endl;
        cout<<"..............................."<<endl;}*/

  double Rxx = 0.5 * Bx * Bx * pow(dz,2);       double Sxx = (1/6)*Bx*Bx*pow(dz,
3);
  double Rxy = 0.5 * Bx * By * pow(dz,2);       double Sxy = (1/6)*Bx*By*pow(dz,
3);
  double Ryx = 0.5 * By * Bx * pow(dz,2);       double Syx = (1/6)*By*Bx*pow(dz,
3);
  double Ryy = 0.5 * By * By * pow(dz,2);       double Syy = (1/6)*By*By*pow(dz,
3);

  double fx = 0.0; double fy = 0.0; double ftx = 0.0; double fty = 0.0;
  fx = -qbyP*Q55*pow(kappa*T,2)*(tx*(3*ty*ty+1)*Sxx - ty*(3*tx*tx+1)*Sxy -ty*(3*
tx*tx+1)*Syx + tx*(3*tx*tx+3)*Syy);
  fy = -qbyP*Q55*pow(kappa*T,2)*(ty*(3*ty*ty+3)*Sxx - tx*(3*ty*ty+1)*Sxy -tx*(3*
ty*ty+1)*Syx + ty*(3*tx*tx+1)*Syy);
  ftx= -qbyP*Q55*pow(kappa*T,2)*(tx*(3*ty*ty+1)*Rxx - ty*(3*tx*tx+1)*Rxy -ty*(3*
tx*tx+1)*Ryx + tx*(3*tx*tx+3)*Ryy);
  fty= -qbyP*Q55*pow(kappa*T,2)*(ty*(3*ty*ty+3)*Rxx - tx*(3*ty*ty+1)*Rxy -tx*(3*
ty*ty+1)*Ryx + ty*(3*tx*tx+1)*Ryy);

  //  double Lo = 0.01757;
  double dl = fabs(T * dz);

  double RdL    = material->GetRadLen();        // cm
  double X0     = 1.e-2 * RdL;                  // m
  double Xs     = X0 * (1.0+1.0/Z) * log(287.0/sqrt(Z))/log(159.0/cbrt(Z));

  {
    //double Cms        = pow((0.0136/(beta * P)),2.0) * (T*fabs(dz)/Lo) * pow(1
```

```
 + 0.038 * log(T*fabs(dz)/Lo),2);
    double Cms  = (225.0 *(1.0+10.0/((double)nHit))* 1.e-6 * fabs(T*dz))/(beta*b
eta*P*P*Xs);    // Fontana-modified
    //double Cms        = (225.0 * 1.e-6 * fabs(T*dz))/(beta*beta*P*P*Xs);
// Fontana
    double Ctxtx= (1 + tx*tx) * T*T * Cms;
    double Ctxty= tx * ty * T*T * Cms;
    double Ctyty= (1 + ty*ty) * T*T * Cms;
    double D;

    if (dz > 0.0) {
      D = + 1.0;
    } else {
      D = - 1.0;
    }
    Q_k_minus[0][0]= Ctxtx * pow(dl,2.0)/3.0;
    Q_k_minus[0][1]= Ctxty * pow(dl,2.0)/3.0;
    Q_k_minus[0][2]= Ctxtx * D * pow(dl,1.0)/2.0;
    Q_k_minus[0][3]= Ctxty * D * pow(dl,1.0)/2.0;
    Q_k_minus[0][4]= kappa * T * (tx*ty*Sx - (1.0+tx*tx)*Sy) * Q55 + fx;

    Q_k_minus[1][0]= Ctxty * pow(dl,2.0)/3.0;
    Q_k_minus[1][1]= Ctyty * pow(dl,2.0)/3.0;
    Q_k_minus[1][2]= Ctxty * D * pow(dl,1.0)/2.0;
    Q_k_minus[1][3]= Ctyty * D * pow(dl,1.0)/2.0;
    Q_k_minus[1][4]= kappa * T * ((1.0+ty*ty)*Sx - tx*ty*Sy) * Q55 + fy;

    Q_k_minus[2][0]= Ctxtx * D * pow(dl,1.0)/2.0;
    Q_k_minus[2][1]= Ctxty * D * pow(dl,1.0)/2.0;
    Q_k_minus[2][2]= Ctxtx;
    Q_k_minus[2][3]= Ctxty;
    Q_k_minus[2][4]= kappa * T * (tx*ty*Rx - (1.0+tx*tx)*Ry) * Q55 + ftx;

    Q_k_minus[3][0]= Ctxty * D * pow(dl,1.0)/2.0;
    Q_k_minus[3][1]= Ctyty * D * pow(dl,1.0)/2.0;
    Q_k_minus[3][2]= Ctxty;
    Q_k_minus[3][3]= Ctyty;
    Q_k_minus[3][4]= kappa * T * ((1.0+ty*ty)*Rx - tx*ty*Ry) * Q55 + fty;

    Q_k_minus[4][0]=kappa * T * (tx*ty*Sx - (1.0+tx*tx)*Sy) * Q55 + fx;
    Q_k_minus[4][1]=kappa * T * ((1.0+ty*ty)*Sx - tx*ty*Sy) * Q55 + fy;
    Q_k_minus[4][2]=kappa * T * (tx*ty*Rx - (1.0+tx*tx)*Ry) * Q55 + ftx;
    Q_k_minus[4][3]=kappa * T * ((1.0+ty*ty)*Rx - tx*ty*Ry) * Q55 + fty;
    Q_k_minus[4][4]=Q55;
  }
  //    cout<<"*****MS matrix ends*****"<<endl;
}
void InoTrackFitAlg::KalmanFilterStateVector(double *x__minus, const int Plane,
const bool GoForward, double x_kk[6]) {
  //cout<<"*****Compare predicted (x,y) with Experimental (x,y) *****"<<endl;

  for (int ij=0; ij<5; ij++) {
    //cout<<"x__minus = "<<x__minus[ij]<<endl;
  }

  //  double H_k[2][5];
  double m_k[2]={0,0};
  double M_k[2]={0,0};
  double sigma_xx = 0;
  double sigma_yy = 0;
  //  double chisquare= 0;

  int PlaneView = TrkClustsData[Plane][0].PlaneView;
  for (int ij=0; ij<2; ij++) {
    for (int jk=0; jk<5; jk++) {
      H_k[ij][jk]=0;
    }
  }
  if (PlaneView%2==0) {
```



```
    H_k[0][0]=1;
    m_k[0]      = TrkClustsData[Plane][0].XPos;
    M_k[0]      = m_k[0];
    sigma_xx= TrkClustsData[Plane][0].XPosErrSq;
  }
  if (PlaneView>0) {
    H_k[1][1]=1;
    m_k[1]      = TrkClustsData[Plane][0].YPos;
    M_k[1]      = m_k[1];
    sigma_yy= TrkClustsData[Plane][0].YPosErrSq;
  }

  for (int ij=0; ij<2; ij++) {
    for (int jk=0; jk<5; jk++) {
      m_k[ij] -= H_k[ij][jk]*x__minus[jk];       //Kisel: m_k[] => [m_k - H_k *
F_(k-1)*a_(k-1)]
    }
  }

  for (int ij = 0; ij < 5; ij++) {
    x_kk[ij] = 0;
  }

  // Calculate x_kk
  for (int ij=0; ij<5; ++ij) {
    x_kk[ij]=x__minus[ij];

    for (int jk=0; jk<2; jk++) {
      x_kk[ij]+=K_k[ij][jk]*m_k[jk];      //x_kk[] => [x_kk + K_k * m_k] // this
 is modefied m_k
    }
  }

  double M_MinusX_k[2]={0};
  M_MinusX_k[0] = M_k[0]-x_kk[0];
  M_MinusX_k[1] = M_k[1]-x_kk[1];

  if(fabs(x_kk[0])<24.5 && fabs(x_kk[1])<8.5) {
    PassTrack=true;
  } else {
    cout<<"================================================"<<endl;
    cout <<M_k[0] <<"  "<<M_k[1] <<endl;
    cout <<x__minus[0] <<"  "<<x__minus[1]<<endl;
    cout <<x_kk[0] <<"  "<<x_kk[1] <<endl;
    cout<<"................................................"<<endl;
    //cout<<"X "<<FilteredData[Plane-2][5].x_k0<<" and Y "<<FilteredData[Plane-2
][5].x_k1<<endl;
    //cout<<"X (Plane-2) "<<TrkClustsData[Plane-2][0].XPos<<" and Y (Plane-2) "<
<TrkClustsData[Plane-2][0].YPos<<endl;

    cout<< " Passtrack 1 at NIter "<<NIter<<endl;
    PassTrack=false;
    cout<<"================================================"<<endl;
  }

  double GPos[3];
  //GPos denotes the starting (x,y,z) of the particle right above the RPC expres
sed in (mm)
  GPos[0]=x_kk[0] * 1000;
  GPos[1]=x_kk[1] * 1000;
  GPos[2]=ZPosLayer[Plane];
  //cout<<"After KF, "<<"GPos[0] = "<<GPos[0]<<"      "<<"GPos[1] = "<<GPos[1]<<"
     "<<"GPos[2] = "<<GPos[2]<<" Plane="<<Plane<<endl;
  if (LastIteration && GoForward ==false && Plane == MinPlane) {
    MinPlaneData[0] = x_kk[0];
    MinPlaneData[1] = x_kk[1];
    MinPlaneData[2] = x_kk[2];
    MinPlaneData[3] = x_kk[3];
    MinPlaneData[4] = x_kk[4];
```

```
    MinPlaneData[5] = GPos[2];
  }
  //cout<<"Save data @ MinPlane"<<endl;
  if (LastIteration && GoForward == true && Plane == MaxPlane) {
    MaxPlaneData[0] = x_kk[0];
    MaxPlaneData[1] = x_kk[1];
    MaxPlaneData[2] = x_kk[2];
    MaxPlaneData[3] = x_kk[3];
    MaxPlaneData[4] = x_kk[4];
    MaxPlaneData[5] = GPos[2];
  }
  //cout<<"Save data @ MaxPlane"<<endl;
}

//Abhijit's Work. ADB 2015/05/06
//>>

int InoTrackFitAlg::CheckFCPCUpOrDn(double *ax_k, bool DirExtraPol, int MaxMinPl
ane, bool GoDir) {

  //bit 1 : LocalPos[1];
  //bit 2 : CheckMat[1]
  //bit 3 : LocalPos[0]
  //bit 4 : CheckMat[0]
  //bit 5-10 : hadron cluster (only for backward)
  //bit 11-12 : Fiducial volume (top and bottom layer)
  //bit 13-16 : Fiducial volume (X/Y side)


  const int nEmpyExtr=2;
  const int nMxShowerlayer=6;
  //  bool tmpFCPC[nEmpyExtr] =0;
  int CheckMat[nEmpyExtr] = {0};
  int LocalPos[nEmpyExtr] = {0};
  int isHadCluster[2] = {0};
  int FiducialZ = 3;
  double FiducialX = 100.0;
  double FiducialY = 100.0;

  double dir[3]={0.0};
  // int N2 = ptrackCollection2->InoTrack_list.size();
  int nextplane =0;
  // double StateVector[6]      = {0.0};
  double Prediction[6]  = {0.0};

  double tmpax_k[5];
  double tmpC_k_intermediate[5][5];
  for (int ij=0; ij<5; ij++) {
    tmpax_k[ij] = ax_k[ij];
    for (int jk=0; jk<5; jk++) {
      tmpC_k_intermediate[ij][jk] = C_k_intermediate[ij][jk];
    }
  }

  //DirExtraPol = 1 for upward extrapolation
  //DirExtraPol = 0 for downward extrapolation

  int iext=0;
  int iempty=0;
  while(iext<nMxShowerlayer && iempty<nEmpyExtr) {
    iext++;
    // cout <<"iext "<<int(DirExtraPol)<<" "<<int(GoDir)<<" "<<iext<<endl;
    double dsExt = 0.0;
    double drangeExt = 0.0;
    if(DirExtraPol) {
      // cout<<"Extrapolating upwards ..."<<endl;
      nextplane = MaxMinPlane + iext;
      if (nextplane >= int(nLayer)) { break;} //{nextplane = nLayer-1;}
    } else {
```



```
      // cout<<"Extrapolating downwards ..."<<endl;
      nextplane = MaxMinPlane - iext;
      if (nextplane <0) { break;} // {nextplane = 0;}
    }

    bool GetPrediction= PredictedStateCov(ax_k, MaxMinPlane, nextplane, GoDir, P
rediction, 0, &dsExt, &drangeExt);

    if (GetPrediction) {
      double pos[3] = {100*Prediction[0],100*Prediction[1],100*ZPosLayer[nextpla
ne]};
      dir[0] = 0.0;
      dir[1] = 0.0;
      dir[2] = 1.0;

      // cout<<"MaxMinPlane = "<<MaxMinPlane<<", nextplane = "<<nextplane<<", Go
Dir = "<<GoDir<<", Prediction = "<<pos[0]<<" "<<pos[1]<<" "<<pos[2]<<", Initial
Pos = "<<100*ax_k[0]<<", "<<100*ax_k[1]<<", "<<100*ZPosLayer[MaxMinPlane]<<endl;

      // cout <<"iext2 "<<iext<<endl;
      icalGeometry->InitTrack(pos, dir);

      double localpos[3];
      icalGeometry->MasterToLocal(pos, localpos);

      localmat= icalGeometry->GetCurrentVolume()->GetMaterial();

      // Putting the conditions to check FC or PC
      CheckMat[iempty] = 0;
      iempty++;
      if (strstr(localmat->GetName(),"rpcgas")) {
        //cout <<"localpos "<<pos[0]<<" "<<pos[1]<<" "<<pos[2]<<" "<<localpos[0]
<<" "<<localpos[1]<<" "<<localpos[2]<<" "<<localmat->GetName()<<endl;

        CheckMat[iempty-1] = 1;
        LocalPos[iempty-1] = 1; // ((pargasxyz[0]-abs(localpos[0])>2.0) &&  (par
gasxyz[1]-abs(localpos[1])>2.0)) ? 1 : 0;
        // cout<<"............................................"<<endl;
        // cout<<"Fully Contained..."<<DirExtraPol<<"..."<<GoDir<<"...NextPlane.
.."<<nextplane<<endl;
        // cout <<"pos "<<pos[0]<<" "<<pos[1]<<" "<<pos[2]<<endl;
        // cout<<"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"<<endl;
        int clusterid = (DirExtraPol!=GoDir) ? 0 : 1;

        // for (unsigned int iht=0; iht<inoHit_pointer->InoHit_list.size(); iht+
+) {
        int nhit=0;
        InoCluster* Clust = 0;
        for (unsigned int iclust=0; iclust<InoCluster_pointer->InoCluster_list.s
ize(); iclust++) {
          if (InoCluster_pointer->InoCluster_list[iclust]->GetZPlane()==nextplan
e) {
            if (abs(InoCluster_pointer->InoCluster_list[iclust]->GetXPos() - Pre
diction[0])<2*StripXWidth &&
                abs(InoCluster_pointer->InoCluster_list[iclust]->GetYPos() - Pre
diction[1])<2*StripYWidth) {
              // cout <<"iclust "<<iclust<<endl;
              for(unsigned int iht=0; iht<InoCluster_pointer->InoCluster_list[ic
lust]->HitsInCluster.size(); iht++) {
                if (abs(InoCluster_pointer->InoCluster_list[iclust]->HitsInClust
er[iht]->GetXPos() - Prediction[0])<2*StripXWidth &&
                    abs(InoCluster_pointer->InoCluster_list[iclust]->HitsInClust
er[iht]->GetYPos() - Prediction[1])<2*StripYWidth) {
                  InoCluster_pointer->InoCluster_list[iclust]->HitsInCluster[iht
]->SetUID(10+clusterid);
                  // InoCluster_pointer->InoCluster_list[iclust]->HitsInCluster[
iht]->SetUID(3+clusterid);
                  nhit++;
                  if(nhit==1) {
```

```
                  Clust = new InoCluster(InoCluster_pointer->InoCluster_list[i
clust]->HitsInCluster[iht]);
                } else if(nhit>1) {
                  Clust->AddHit(InoCluster_pointer->InoCluster_list[iclust]->H
itsInCluster[iht]);
                }
                // inoHit_pointer->InoHit_list[iht]->SetUID(3+clusterid);
                LocalPos[iempty-1] = 0;
                CheckMat[iempty-1] = 0;
                iempty = 0;
                // cout<<"Cluster Found....."<<endl;
                // cout<<"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"<<end
l;
              }
            }
          }
        }
      }

      // cout <<"nhit "<<nhit<<endl;
      if(!GoDir) {
        if (nhit>isHadCluster[1]) { isHadCluster[1] =nhit;}
        if (nhit>isHadCluster[0]) { isHadCluster[1] = isHadCluster[0]; isHadCl
uster[0]=nhit;}

        FiducialZ = min( min( FiducialZ, nextplane), int(nLayer-1-nextplane));
        FiducialX = min(FiducialX, abs(IcalX-abs(Prediction[0])));
        FiducialY = min(FiducialY, abs(IcalY-abs(Prediction[1])));

        for (int ix=0; ix<5; ix++) {
          ax_k[ix] = Prediction[ix];
        }
        ExtrapCovMatrixall(); // Also propagate the error matrix
        // cout <<"GPL "<<GPL<<endl;
        if (Clust) {
          fTrackCand->ClustsInTrack.push_back(Clust);
          GPL += dsExt;
          RNG += drangeExt;
          if (DirExtraPol) {
            MaxPlane = nextplane;
          } else {
            MinPlane = nextplane;
          }

          //Update track paramters in local variables
          for (int ij=0; ij<5; ij++) {
            tmpax_k[ij] = ax_k[ij];

            FiltDataStruct temp;

            temp.x_k0=ax_k[0]; temp.x_k1=ax_k[1];
            temp.x_k2=ax_k[2]; temp.x_k3=ax_k[3];
            temp.x_k4=ax_k[4];

            temp.x_k5=1;
            temp.x_k6=true;
            FilteredData[nextplane].push_back(temp);

            for (int jk=0; jk<5; jk++) {
              tmpC_k_intermediate[ij][jk] = C_k_intermediate[ij][jk];
            }
          }
        }

      }
      // cout <<"nhit2 "<<nhit<<endl;
    } //  if (strstr(localmat->GetName(),"rpcgas"))
    //      tmpFCPC = CheckMat && LocalPos;
    // cout<<"Dir "<<DirExtraPol<<", FCorPC "<<FiducialX<<" "<<FiducialY<<endl
```



```
;
    } //  for(int r=iext;r<iext; iext++) {
    // cout<<"yyyyy "<<endl;
  }

  //return back in global track paramter
  for (int ij=0; ij<5; ij++) {
    ax_k[ij] = tmpax_k[ij];
    for (int jk=0; jk<5; jk++) {
      C_k_intermediate[ij][jk] = tmpC_k_intermediate[ij][jk];
    }
  }

  int alltags = 0;
  if (!GoDir) {
    int tmpx = min(3, int(FiducialX*10)); //FiducialX is in metre and want to ha
ve scale of 10cm
    int tmpy = min(3, int(FiducialY*10));
    alltags +=tmpx;
    alltags<<=2;
    alltags +=tmpy;
    alltags<<=2;
    alltags +=FiducialZ;
    alltags<<=3;
    alltags +=min(7, isHadCluster[0]);
    alltags<<=3;
    alltags +=min(7, isHadCluster[1]);
    alltags<<=4;
  }
  alltags +=8*CheckMat[0]+4*LocalPos[0]+2*CheckMat[1]+LocalPos[1];
  // cout <<"alltags "<<alltags<<endl;
  return alltags;

}

bool InoTrackFitAlg::CheckFCPC(double *ax_k, bool GoForward) {
  int ExtremePlane = 0;
  ExtremePlane = (GoForward == true) ? MaxPlane:MinPlane;

  double epsilon  = 1.e-6;
  double dz             = 0.000000000000000000000000000000000000000;
  double signp  = (GoForward == true) ? 1.0000 : -1.0000000;
  double costhinv = sqrt(1.0 + ax_k[2]*ax_k[2] + ax_k[3]*ax_k[3]);
  costhinv              = (GoForward == true) ? costhinv :-costhinv;

  double pos[3]={0.0};  pos[0] = 1.e3*ax_k[0];  pos[1] = 1.e3*ax_k[1];  pos[2] =
 1.e3*ZPosLayer[ExtremePlane];
  double dir[3]={0.0};  dir[0] = ax_k[2]/costhinv;      dir[0] = ax_k[3]/costhin
v;      dir[0] = 1.0/costhinv;

  icalGeometry->InitTrack(pos, dir);

  double bx = 0.0; double by = 0.0;
  pFieldMap->ElectroMagneticField(pos,bx,by,1);
  bx *= 1000;
  by *= 1000;

  //  TGeoMaterial *mat= icalGeometry->GetCurrentVolume()->GetMaterial();
  icalGeometry->FindNextBoundary();
  double snext= epsilon+gGeoManager->GetStep();
  //  double atNum= mat->GetZ();

  if (snext > 1.0) snext=1.0;

  if (strstr(icalGeometry->GetCurrentVolume()->GetName(),"IRLAYElog")) {
    dz = signp*min(0.001, 0.01*snext*abs(dir[2]));
  } else {
    bx = by = 0.0;
```

```
    dz = 0.01*signp*snext*fabs(dir[2]); // cm to meter
    if(strstr(icalGeometry->GetCurrentVolume()->GetName(),"GASRlog")) { dz = 0.0
01*signp;}
  }

  double x = ax_k[0];   double y = ax_k[1];     double tx = ax_k[2];    double t
y = ax_k[3];//  double qp = ax_k[4];
  double projection[5]={0.0};
  double qbyP = ax_k[4];// 0;
  double kappa  = 0.299792458;
  double T              = sqrt(1+pow(tx,2)+pow(ty,2));
  if (ZIncreasesWithTime==false) T=-T;
  double h              = kappa*qbyP*T;
  double Rx             = bx*dz*h;
  double Ry             = by*dz*h;
  double Sx             = 0.5*bx*pow(dz,2)*h;
  double Sy             = 0.5*by*pow(dz,2)*h;
  double Rxx            = 0.5 * bx * bx * pow(dz,2);    double Sxx = (1/6)*bx*bx
*pow(dz,3);
  double Rxy            = 0.5 * bx * by * pow(dz,2);    double Sxy = (1/6)*bx*by
*pow(dz,3);
  double Ryx            = 0.5 * by * bx * pow(dz,2);    double Syx = (1/6)*by*bx
*pow(dz,3);
  double Ryy            = 0.5 * by * by * pow(dz,2);    double Syy = (1/6)*by*by
*pow(dz,3);

  projection[0] = x + tx * dz + tx * ty * Sx - (pow(tx,2)+1) * Sy + h*h * (tx*(3
*ty*ty+1)*Sxx - ty*(3*tx*tx+1)*Sxy -ty*(3*tx*tx+1)*Syx + tx*(3*tx*tx+3)*Syy);
  projection[1] = y + ty * dz + (pow(ty,2)+1) * Sx - tx * ty * Sy + h*h * (ty*(3
*ty*ty+3)*Sxx - tx*(3*ty*ty+1)*Sxy -tx*(3*ty*ty+1)*Syx + ty*(3*tx*tx+1)*Syy);
  projection[2] = tx + tx * ty * Rx - (pow(tx,2)+1) * Ry + h*h * (tx*(3*ty*ty+1)
*Rxx - ty*(3*tx*tx+1)*Rxy -ty*(3*tx*tx+1)*Ryx + tx*(3*tx*tx+3)*Ryy);
  projection[3] = ty + (pow(ty,2)+1) * Rx - tx * ty * Ry + h*h * (ty*(3*ty*ty+3)
*Rxx - tx*(3*ty*ty+1)*Rxy -tx*(3*ty*ty+1)*Ryx + ty*(3*tx*tx+1)*Ryy);

  return true;
}
//<<
//Abhijit's Work ADB 2015/05/06


/*
   bool InoTrackFitAlg::CheckSign(double x, double y, double tx, double ty) {
   bool sign;
   double dz;
   double TT = sqrt(1 + tx*tx + ty*ty);

   if (ZIncreasesWithTime==true) {
   signp = (ZIncreasesWithTime != GoForward) ? -1 :+1;vtxz=MinPlane;endz=MaxPlan
e;
   } else {
   signp = (ZIncreasesWithTime != GoForward) ? +1 :-1;vtxz=MaxPlane;endz=MinPlan
e;
   }

   double Vtx[3]={x, y, vtxz};
   double Dir[3]=signp*{(tx/TT), (ty/TT), (1.0/TT)};

   for (int ij = 0; ij<nHit; ij++) {
   icalGeometry->InitTrack(Vtx, Dir);
   Bx = By = 0.0;
   pFieldMap->ElectroMagneticField(MPos,Bx,By,1);
   Bx *= 1000;
   By *= 1000;

   localmat= icalGeometry->GetCurrentVolume()->GetMaterial();
   icalGeometry->FindNextBoundary();
   snext        = epsilon+gGeoManager->GetStep();
   Z            = localmat->GetZ();
```



```
   if (snext > 1.0) snext=1.0;

   if (strstr(icalGeometry->GetCurrentVolume()->GetName(),"IRLAYElog")) {
   dz = signp*min(0.001, 0.01*snext*abs(dirTrack[2]));
   } else {
   Bx = 0.0; By = 0.0;
   dz = 0.01*signp*snext*abs(dirTrack[2]); // cm to meter
   if(strstr(icalGeometry->GetCurrentVolume()->GetName(),"GASRlog"))
   dz = 0.001*signp;
   }
   }

   return sign;
   }
*/
```