Software tutorial: framework session

A. Sarti

Dipartimento di Scienze di Base e Applicate per l'Ingegneria

Sapienza Università di Roma

Framework: why?

- FOOT is a big particle physics experiment, composed of many subsystems
- The full event reconstruction proceeds in steps: starting from the decoding of the data/MC 'raw' information the different sub-detectors process the information and 'complex' objects are built in a sequence of actions.. Eg in the VTX detector:
  - pixels are combined in clusters
  - clusters in tracks
  - tracks are used to form a vertex…..
  - starting from the tracks associated to a given vertex a full event is reconstructed 'forward-projecting' the tracks in the other detectors (IT, MSD, TW, Calo…)

Framework: how?

- To provide an 'easy' access to a) input data processed through a b) chain of actions producing c) output data the ROOT framework has been chosen
- You 'only' need to define
  - Which object you want to create, what input is needed and how you want to create it.
  - ROOT does the rest: 'automagically' organises the actions that are needed, in proper order, and
    - Pre-configures what is needed (if needed)
    - Executes, on an event-by-event basis, the actions
    - runs a post-configuration action (if defined)
  - The information about the input and output objects is 'updated' and 'cleared' for each event

Parameters…

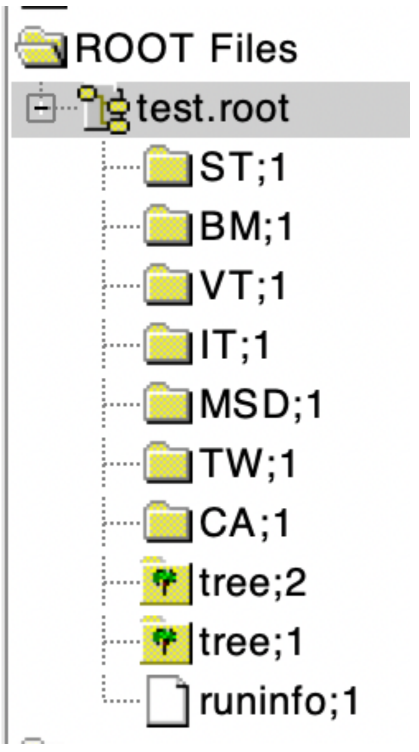
- Beside what changes on an event-by event basis, there are some informations that are needed to 'reconstruct' the objects, but that are not changing in each event. There are the 'parameters'.
- Eg: to reconstruct a 'track' starting from the information of a 'pixel' detector one needs to know the 3D position of each pixel ("geometry")… Other examples of useful information: the 'configuration' of a detector, its 'calibration' and the mapping of the readout channels…
  - All these are 'TAGparaDsc' objects.
- Together with the 'data' [TAGdataDsc] and the actions [TAGaction] the parameters are completing everything that is done in shoe.
  - All the rest is 'examples' :) … ok ok .. I'm over-simplifying a little bit.

The simplest job…

- main chain:
  - First of all you need to define ‘the input’ (usually a ROOT file) and what you want to use out of it
  - Then you pre-load all the information that is not changing ‘event by event’, define the objects that you want to build, and the actions that are needed for doing so
  - Then you execute the actions ‘pre-configuration’ step
  - You process all the events and execute the post-configuration step
  - You close the output files, and it’s done!
- Disclaimer: it’s not trivial or easy. E.g. you’ll see that in ROOT the ‘order’ in which you do things matters!
  - Eg. you need to define the objects you want to read before opening the file that contains it, or you’ll end up with a lot of random numbers :)

Reverse engineering! (I)

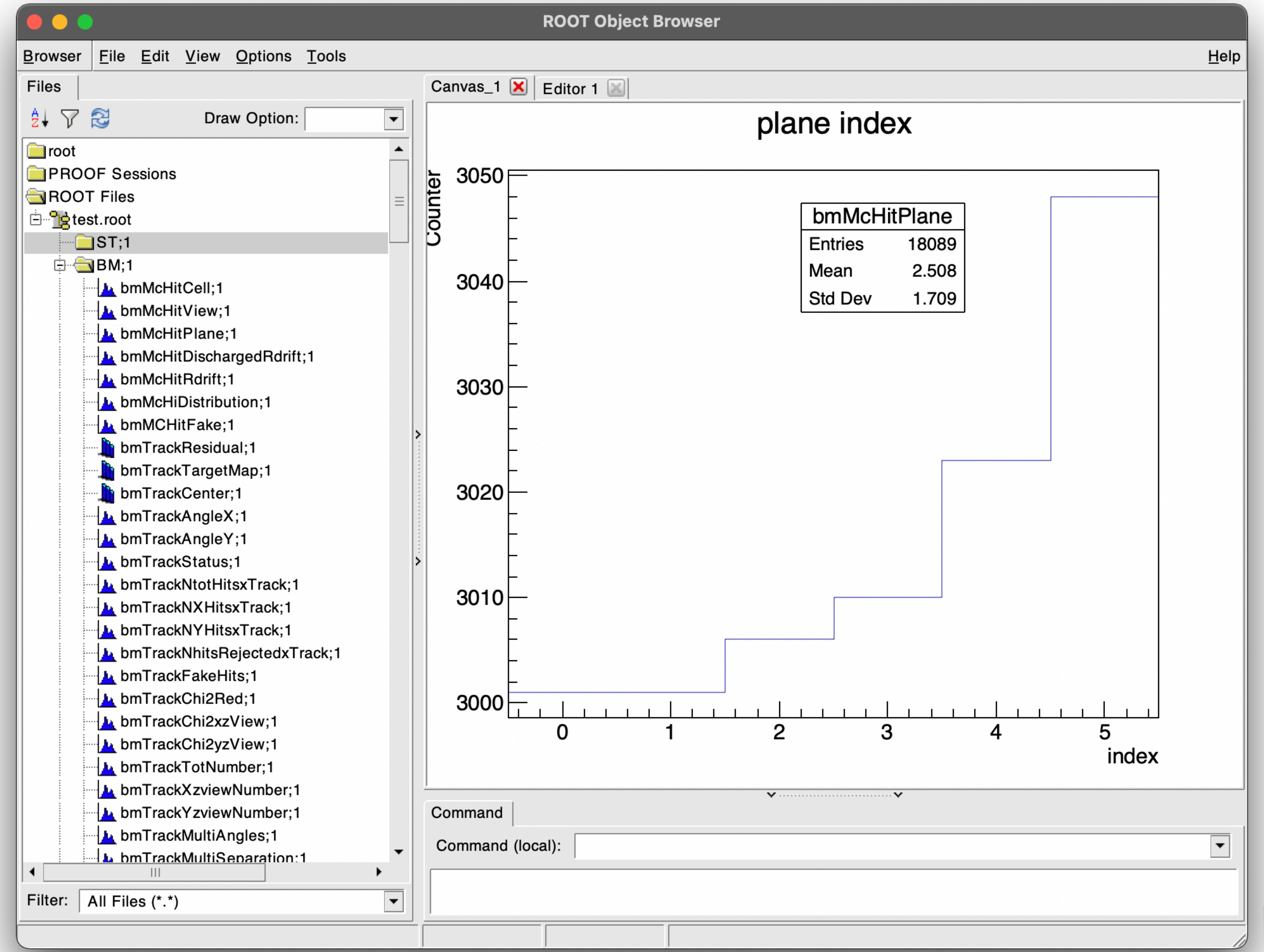
- Let's try to understand how things are done in shoe by starting .. from the output!
- Imagine **that someone has already used shoe to process some MC or data collection of events** (../../bin/DecodeMC -in Tutorial/Full/12C_C_200shoe.root -out test.root -nev 1500 -exp 12C_200 -run 1).
- The output will be a ROOT file containing three main objects useful for your investigations:
  - An ntuple, a run info object, a collection of histograms
- Let's see how we can understand who produced what and how we can put our hands on the algorithms and 'tune/change' them to change the info or add some info according to our needs…



```
ROOT Files
  test.root
    ST;1
    BM;1
    VT;1
    IT;1
    MSD;1
    TW;1
    CA;1
    tree;2
    tree;1
    runinfo;1
```

The ‘easy’ part: histograms

- E.g. I’d like to understand what I can ‘get’ about BM reconstruction.
- there are plenty of histograms: how are built? filled? saved?
- E.g. bmMcHitPlane: how can I find info about it?



root
PROOF Sessions
ROOT Files
test.root
ST;1
BM;1
bmMcHitCell;1
bmMcHitView;1
bmMcHitPlane;1
bmMcHitDischargedRdrift;1
bmMcHitRdrift;1
bmMcHiDistribution;1
bmMCHitFake;1
bmTrackResidual;1
bmTrackTargetMap;1
bmTrackCenter;1
bmTrackAngleX;1
bmTrackAngleY;1
bmTrackStatus;1
bmTrackNtotHitsxTrack;1
bmTrackNXHitsxTrack;1
bmTrackNYHitsxTrack;1
bmTrackNhitsRejectedxTrack;1
bmTrackFakeHits;1
bmTrackChi2Red;1
bmTrackChi2xzView;1
bmTrackChi2yzView;1
bmTrackTotNumber;1
bmTrackXzviewNumber;1
bmTrackYzviewNumber;1
bmTrackMultiAngles;1
bmTrackMultiSeparation;1

bmMcHitPlane	
Entries	18089
Mean	2.508
Std Dev	1.709

Reverse engineering: grep (II)

- Beside the 'google' interpretation of GIYF, you'll learn that also 'grep' is your friend!
  - shoe has MANY classes, packages, tons of lines of code. How can I understand where my favourite code is kept?
    - 'names' should help :) … Eg. if I'm looking for BM - related info, a good place to start is the TABMbase package.
    - however from time to time there are algorithms that are shared btw different subsystems (e.g. the decoding of the WaveDream output is shared among SC, TW and CALO)… what can I do?
- G(rep)IFY!
  - grep bmMcHitPlane libs/src/TA*/* will try to look for the 'bmMcHitPlane' string inside the base classes of shoe located inside libs/src.
  - Issued from shoe one gets: grep bmMcHitPlane libs/src/TA*/*
  - **libs/src/TABMbase/TABMactNtuHitMC.cxx**: fpHisPlane = new TH1I("bmMcHitPlane", "plane index; index; Counter", 6, -0.5, 5.5);
  - **libs/src/TABMbase/TABMactNtuMC.cxx**: fpHisPlane = new TH1I("bmMcHitPlane", "plane index; index; Counter", 6, -0.5, 5.5);

Reverse engineering: grep (III)

- Good! We found what we need and we can get already some interesting info:
  - libs/src/TABMbase/TABMactNtuHitMC.cxx: fpHisPlane = new TH1I("bmMcHitPlane", "plane index; index; Counter", 6, -0.5, 5.5);
  - libs/src/TABMbase/TABMactNtuMC.cxx: fpHisPlane = new TH1I("bmMcHitPlane", "plane index; index; Counter", 6, -0.5, 5.5);
- The histograms are defined inside an action (TABM**act**Ntu*) that is executed on an 'event by event basis' [as expected]. The line of code we found is related to the declaration of the hits (new TH1I(….)) and to see how the histo is used, we need to look inside the file and search for fpHisPlane!
  - Before understanding how an action works, a tougher task: there are two actions… How can I understand which one was used to create the histo I am looking at right now???

Reverse engineering: grep (IV)

- which one?
  - libs/src/TABMbase/TABMactNtuHitMC.cxx: fpHisPlane = new TH1I("bmMcHitPlane", "plane index; index; Counter", 6, -0.5, 5.5);
  - libs/src/TABMbase/TABMactNtuMC.cxx: fpHisPlane = new TH1I("bmMcHitPlane", "plane index; index; Counter", 6, -0.5, 5.5);
- Depends! If there are two actions, it means that both are doing specific things.. that can be used by the user to perform specific actions. Which one was used creating the output?
  - If you are using a macro.. it's easy! Look inside the macro and see which action you're calling :)
  - If you are using the shoe executables.. Not easy to answer! That is the hardest part you'll encounter… You need to know a little bit about shoe executables to understand who is called.. But, once again… GIYF.

Reverse engineering: grep (V)

- which one?
  - libs/src/TABMbase/TABMactNtuHitMC.cxx or libs/src/TABMbase/TABMactNtuMC.cxx
- The classes used to code the executables are kept inside TAGfoot. Let's try grep once again:
  - If you are using a macro.. it's easy! Look inside the macro and see which action you're calling :)
  - If you are using the shoe executables.. Not easy to answer! That is the hardest part you'll encounter… You need to know a little bit about shoe executables to understand who is called.. But, once again… GIYF.
  - From TAGfoot: grep TABMactNtuHitMC *
    - LocalRecoNtuMC.cxx: fActNtuRawBm = new TABMactNtuHitMC("bmActNtu", fpNtuMcBm, fpNtuMcEve, fpNtuRawBm, fpParConfBm, fpParCalBm, fpParGeoBm, fEvtStruct);
  - From TAGfoot: grep TABMactNtuMC *
    - LocalRecoMC.cxx: fActNtuRawBm = new TABMactNtuMC("bmActNtu", fpNtuRawBm, fpParConfBm, fpParCalBm, fpParGeoBm, fEvtStruct);

Executables in shoe!

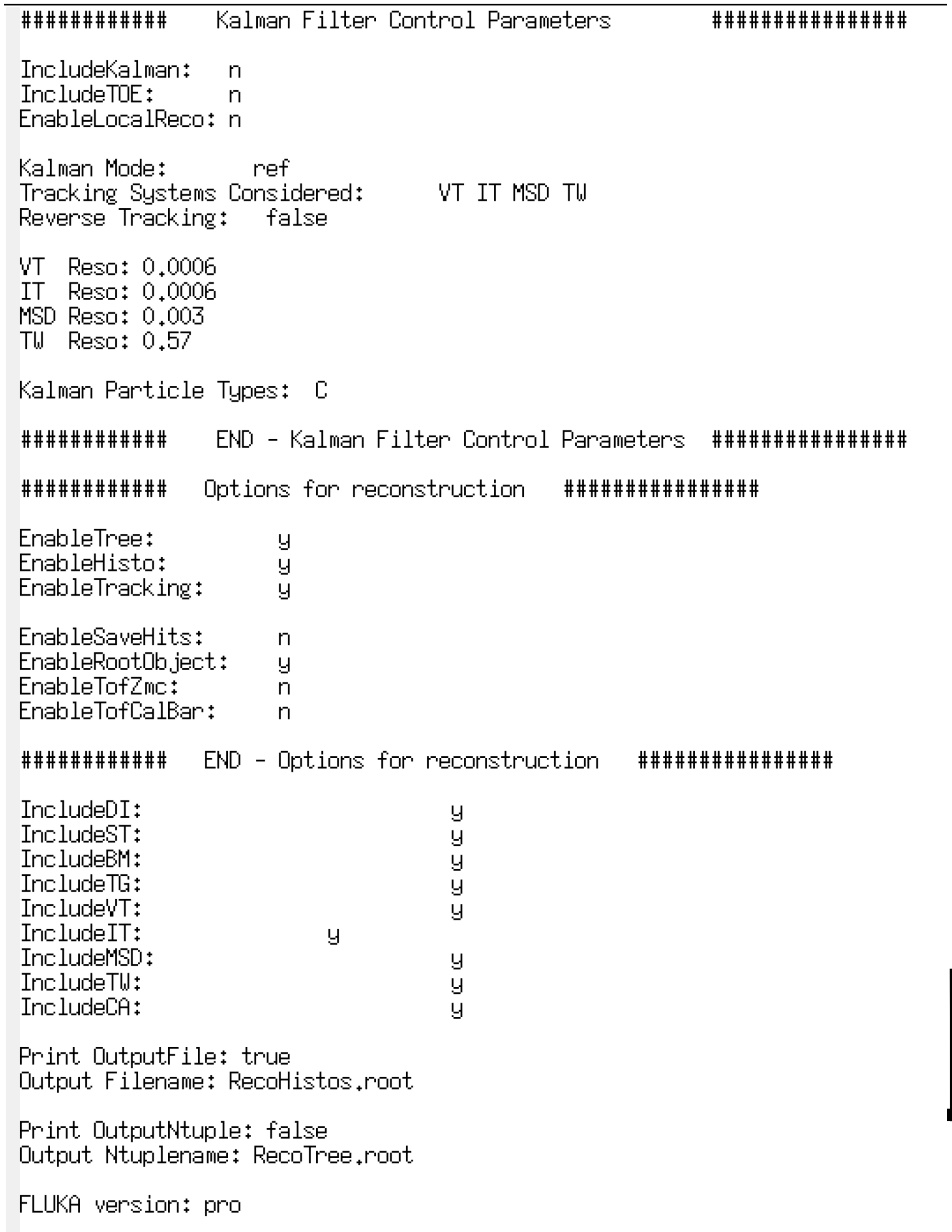
- which one?
  - libs/src/TABMbase/TABMactNtuHitMC.cxx or libs/src/TABMbase/TABMactNtuMC.cxx
- This means that the real choice is btw:
  - LocalRecoNtuMC.cxx and LocalRecoMC.cxx.. Once again using grep we find out that
    - The class is directly implemented inside the executables inside Reconstruction/level0
    - DecodeMC contains both:
    - if (!obj && !test)   locRec = new LocalRecoMC(exp, runNb, in, out);
    - else   locRec = new LocalRecoNtuMC(exp, runNb, in, out);
  - So: the user can decide wether (s)he wants to process root objects as input (obj flag) or if, instead, as input expects a simple 'ROOT tree'.

An example: DecodeMC

- Depending on what you need to do, there are already several options available at your hand…
  - You want to process MC simulation output from FLUKA? DecodeMC is what you're looking for.
  - If you are looking at 'old' tuples, with the old structure format, (e.g. Full/12C_C_200.root file) you should use the LocalRecoMC class, otherwise if you are processing the new rootfiles (e.g. Full/12C_C_200shoe.root file) in which the info is already tupled using the shoe framework and objects, you should use LocalRecoNtuMC.
- How to tell your executable what you want to do?
  - There's a configuration file for it [config/12C_200/FootGlobal.par]. And you should spend some time trying to understand what each line of it is doing..
  - The one we are interested in right now is:
    - EnableRootObject: n
    - [default is 'n': old root format.. to process 'shoe' files you should put it to 'y']

The config file…

- Configuration files are kept under config/XXX/*
  - More info on XXX will come later in a session dedicated to the CampaignManager.
  - Eg. of XXX are: CNAO2020, 12C_200, etc etc
- easy flags:
  - IncludeXX: include a specific subsystem
  - Enable* : almost self-explaining.. tree (saves the ntuple) histo (saves the histograms)…



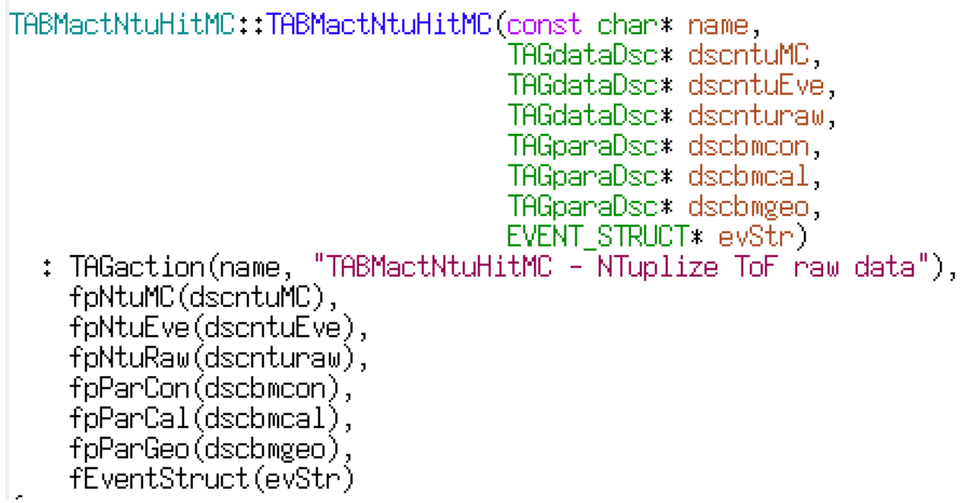
```
############    Kalman Filter Control Parameters        ################

IncludeKalman:   n
IncludeTOE:      n
EnableLocalReco: n

Kalman Mode:        ref
Tracking Systems Considered:      VT IT MSD TW
Reverse Tracking:    false

VT  Reso: 0.0006
IT  Reso: 0.0006
MSD Reso: 0.003
TW  Reso: 0.57

Kalman Particle Types:  C

############    END - Kalman Filter Control Parameters    ################

############   Options for reconstruction    ################

EnableTree:           y
EnableHisto:          y
EnableTracking:       y

EnableSaveHits:       n
EnableRootObject:     y
EnableTofZmc:         n
EnableTofCalBar:      n

############   END - Options for reconstruction    ################

IncludeDI:                          y
IncludeST:                          y
IncludeBM:                          y
IncludeTG:                          y
IncludeVT:                          y
IncludeIT:                y
IncludeMSD:                         y
IncludeTW:                          y
IncludeCA:                          y

Print OutputFile: true
Output Filename: RecoHistos.root

Print OutputNtuple: false
Output Ntuplename: RecoTree.root

FLUKA version: pro
```

Back to histograms!

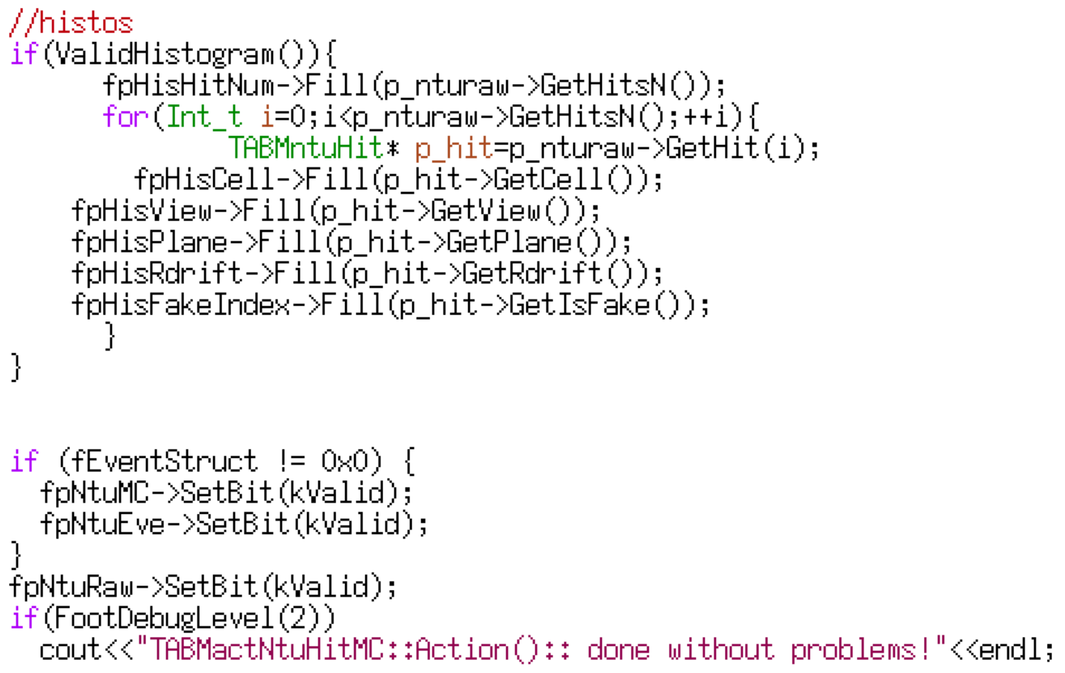
- Now that we have understood who is creating the histogram… Let's have a look at the action! (we use as example TABMactNtuHitMC assuming that you are processing the 'shoe' MC files, -obj flag on)
  - the actions comes with a CreateHistogram() where the histo are created/ booked. This method is called inside the TAGfoot package, BaseReco class, if the histogram output is enabled [EnableHisto: y line in the config file]
  - the fpHisPlane pointer is created and used inside the 'action':
    - Inside TABMactNtuHitMC::Action() [method executed on an event-by-event basis] we call:
    - fpHisPlane->Fill(p_hit->GetPlane());
    - that takes the 'plane' info from MC and stores it inside the histogram.
- Take home message:
  - If I want to change anything in the histo, I go to TABMactNtuHitMC, change what I need and recompile/rerun…!

Action ex. : TABMactNtuHitMC

- The action defines the input data (ntuMC and ntuEve) the out data (ntuRaw) the 'conditions' (config, calibration, geometry)…
- The action 'fills' the histograms (ValidHistogram() is needed to verify that the user wants to save the histo information)
- The 'logging' is handled by FootDebugLevel()… more on that later..
- The action 'ends' declaring that the 'out' data is fine ntuRaw->SetBit(kValid)



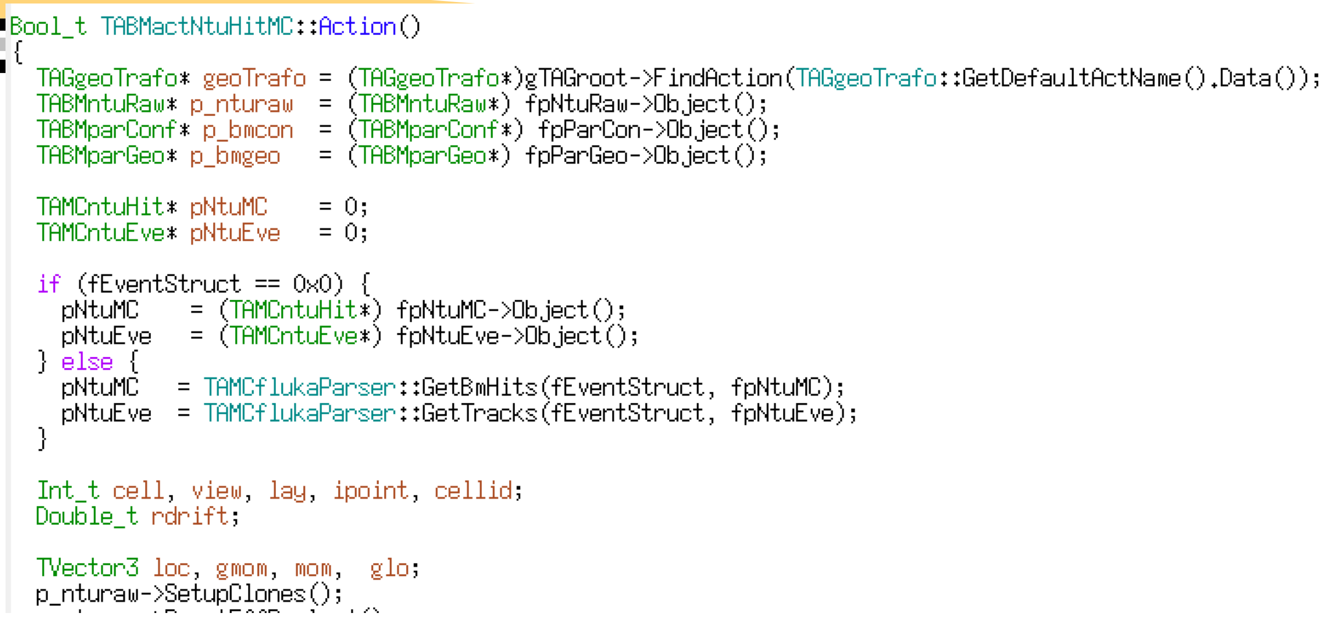
```
TABMactNtuHitMC::TABMactNtuHitMC(const char* name,
                                 TAGdataDsc* dscntuMC,
                                 TAGdataDsc* dscntuEve,
                                 TAGdataDsc* dscnturaw,
                                 TAGparaDsc* dscbmcon,
                                 TAGparaDsc* dscbmcal,
                                 TAGparaDsc* dscbmgeo,
                                 EVENT_STRUCT* evStr)
  : TAGaction(name, "TABMactNtuHitMC - NTuplize ToF raw data"),
    fpNtuMC(dscntuMC),
    fpNtuEve(dscntuEve),
    fpNtuRaw(dscnturaw),
    fpParCon(dscbmcon),
    fpParCal(dscbmcal),
    fpParGeo(dscbmgeo),
    fEventStruct(evStr)
```



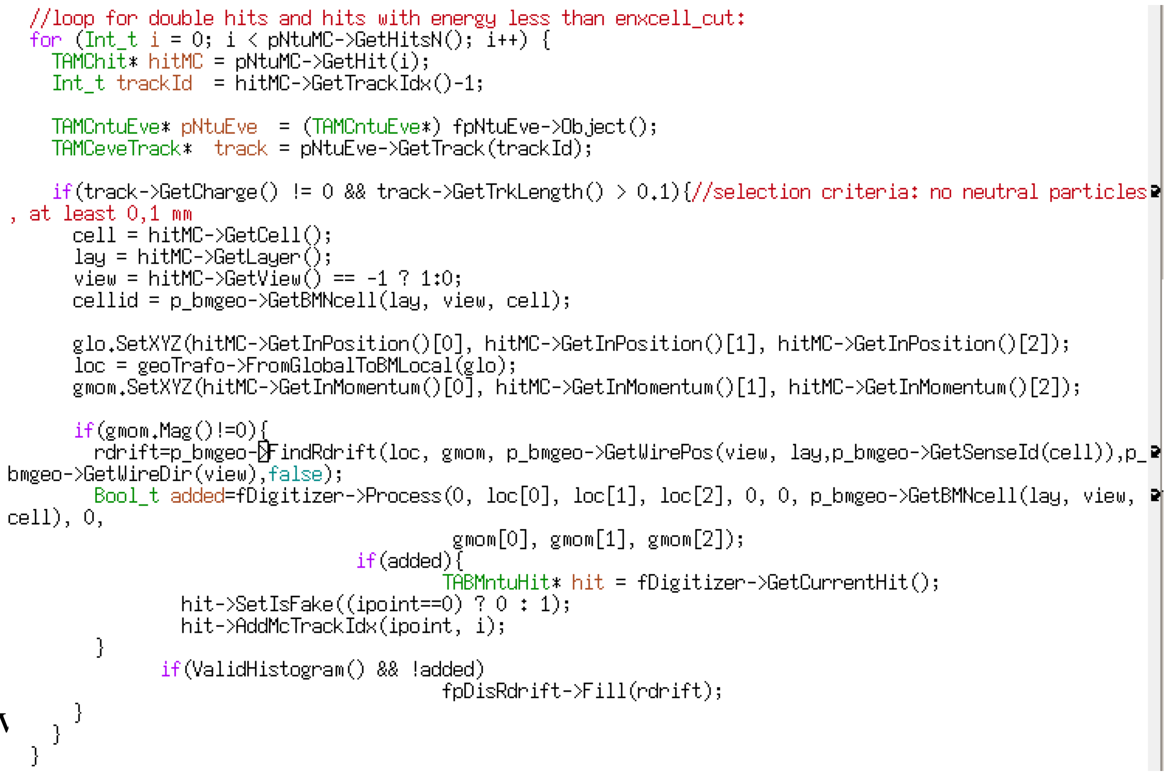
```
//histos
if(ValidHistogram()){
      fpHisHitNum->Fill(p_nturaw->GetHitsN());
      for(Int_t i=0;i<p_nturaw->GetHitsN();++i){
              TABMntuHit* p_hit=p_nturaw->GetHit(i);
        fpHisCell->Fill(p_hit->GetCell());
    fpHisView->Fill(p_hit->GetView());
    fpHisPlane->Fill(p_hit->GetPlane());
    fpHisRdrift->Fill(p_hit->GetRdrift());
    fpHisFakeIndex->Fill(p_hit->GetIsFake());
      }
}


if (fEventStruct != 0x0) {
  fpNtuMC->SetBit(kValid);
  fpNtuEve->SetBit(kValid);
}
fpNtuRaw->SetBit(kValid);
if(FootDebugLevel(2))
  cout<<"TABMactNtuHitMC::Action():: done without problems!"<<endl;
```

Action ex. : TABMactNtuHitMC

- The global geometry is handled by geografo, while the local one from TABMparGeo.
- Event by event one
  - ‘initialise’ the data (SetupClones)
  - use the config/geo information
  - creates and fill a new ‘BM’ hit (digitizer->Process())
  - fills the histograms



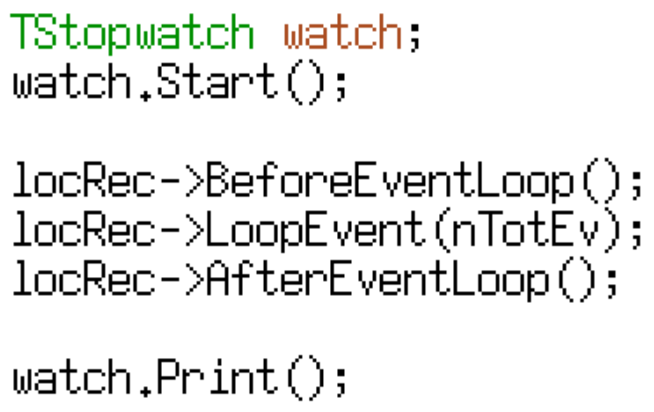
```
Bool_t TABMactNtuHitMC::Action()
{
  TAGgeoTrafo* geoTrafo = (TAGgeoTrafo*)gTAGroot->FindAction(TAGgeoTrafo::GetDefaultActName().Data());
  TABMntuRaw* p_nturaw  = (TABMntuRaw*) fpNtuRaw->Object();
  TABMparConf* p_bmcon  = (TABMparConf*) fpParCon->Object();
  TABMparGeo* p_bmgeo   = (TABMparGeo*) fpParGeo->Object();

  TAMCntuHit* pNtuMC    = 0;
  TAMCntuEve* pNtuEve   = 0;

  if (fEventStruct == 0x0) {
    pNtuMC     = (TAMCntuHit*) fpNtuMC->Object();
    pNtuEve    = (TAMCntuEve*) fpNtuEve->Object();
  } else {
    pNtuMC    = TAMCflukaParser::GetBmHits(fEventStruct, fpNtuMC);
    pNtuEve   = TAMCflukaParser::GetTracks(fEventStruct, fpNtuEve);
  }

  Int_t cell, view, lay, ipoint, cellid;
  Double_t rdrift;

  TVector3 loc, gmom, mom,  glo;
  p_nturaw->SetupClones();
```



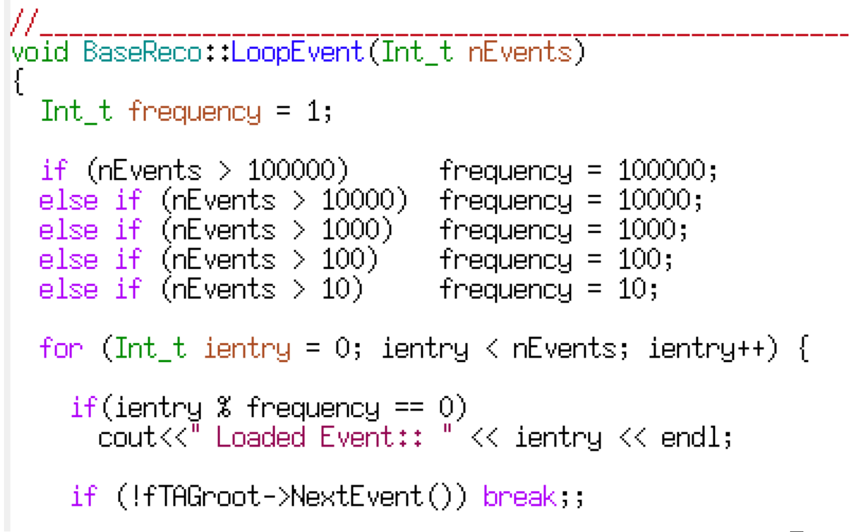
```
      //loop for double hits and hits with energy less than enxcell_cut:
      for (Int_t i = 0; i < pNtuMC->GetHitsN(); i++) {
        TAMChit* hitMC = pNtuMC->GetHit(i);
        Int_t trackId  = hitMC->GetTrackIdx()-1;

        TAMCntuEve* pNtuEve  = (TAMCntuEve*) fpNtuEve->Object();
        TAMCeveTrack*  track = pNtuEve->GetTrack(trackId);

        if(track->GetCharge() != 0 && track->GetTrkLength() > 0.1){//selection criteria: no neutral particles
    , at least 0,1 mm
          cell = hitMC->GetCell();
          lay = hitMC->GetLayer();
          view = hitMC->GetView() == -1 ? 1:0;
          cellid = p_bmgeo->GetBMNcell(lay, view, cell);

          glo.SetXYZ(hitMC->GetInPosition()[0], hitMC->GetInPosition()[1], hitMC->GetInPosition()[2]);
          loc = geoTrafo->FromGlobalToBMLocal(glo);
          gmom.SetXYZ(hitMC->GetInMomentum()[0], hitMC->GetInMomentum()[1], hitMC->GetInMomentum()[2]);

          if(gmom.Mag()!=0){
            rdrift=p_bmgeo->FindRdrift(loc, gmom, p_bmgeo->GetWirePos(view, lay,p_bmgeo->GetSenseId(cell)),p_
    bmgeo->GetWireDir(view),false);
            Bool_t added=fDigitizer->Process(0, loc[0], loc[1], loc[2], 0, 0, p_bmgeo->GetBMNcell(lay, view,
    cell), 0,
                                              gmom[0], gmom[1], gmom[2]);
                                    if(added){
                                            TABMntuHit* hit = fDigitizer->GetCurrentHit();
                    hit->SetIsFake((ipoint==0) ? 0 : 1);
                    hit->AddMcTrackIdx(ipoint, i);
            }
                  if(ValidHistogram() && !added)
                                            fpDisRdrift->Fill(rdrift);
          }
        }
      }
```

Before and after loops..

- The framework structure is the following:
  - After the 'initialisation phase' you have three steps beforeEL, loop, afterEL.
- Coded inside BaseReco.cxx inside TAGfoot. Go and have a look! :)
  - beforeEL loads the geometry, calibration, configuration files.. and creates all the objects needed during the loop and the related actions. It also handles the 'reading' of the input file.
  - LoopEvent is quite simple! It 'only' calls 'NextEvent()' the method that triggers the sequential running of all the Actions() defined in before EL



```
TStopwatch watch;
watch.Start();

locRec->BeforeEventLoop();
locRec->LoopEvent(nTotEv);
locRec->AfterEventLoop();

watch.Print();
```



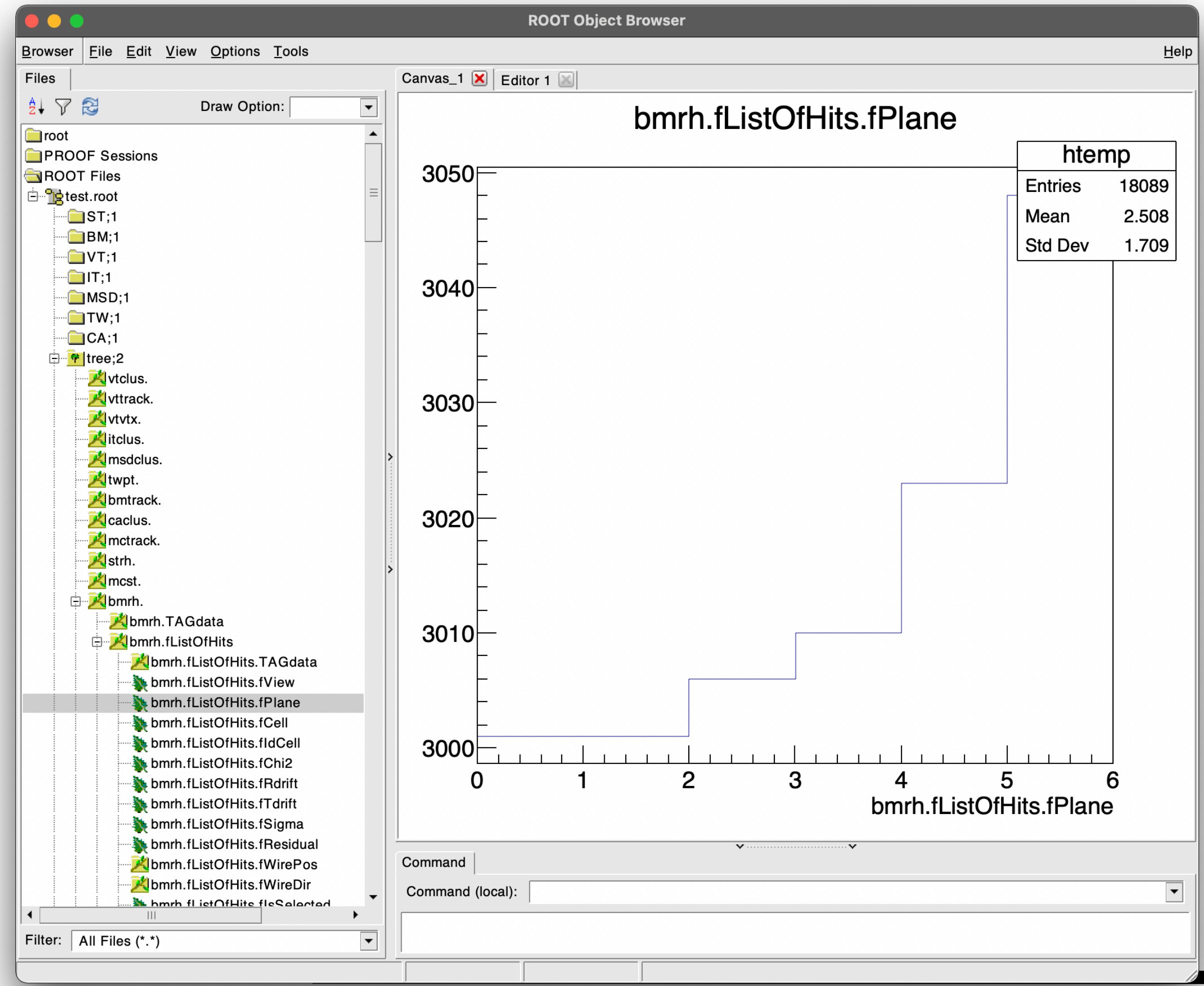
```
//______________________________________________
void BaseReco::LoopEvent(Int_t nEvents)
{
  Int_t frequency = 1;

  if  (nEvents > 100000)      frequency = 100000;
  else if (nEvents > 10000)   frequency = 10000;
  else if (nEvents > 1000)    frequency = 1000;
  else if (nEvents > 100)     frequency = 100;
  else if (nEvents > 10)      frequency = 10;

  for (Int_t ientry = 0; ientry < nEvents; ientry++) {

    if(ientry % frequency == 0)
      cout<<" Loaded Event:: " << ientry << endl;

    if (!fTAGroot->NextEvent()) break;;
```

More interesting: objects!

- Beside histograms one has also access to a ntuple.. That contains the full information.. But in a 'shoe' format :)

- Most information 'is clickable' … for quick plotting.. but not all of it.

- Getting access to the full info from the tuple, is a 'tough' job.. we will see it in detail..

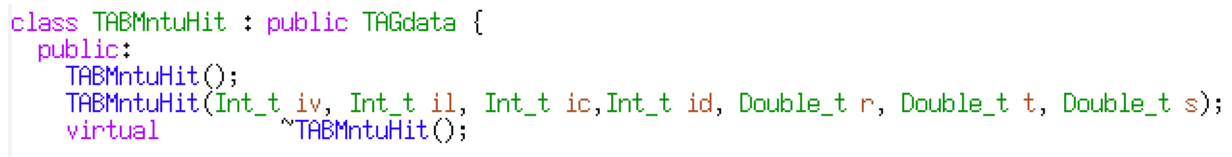


Reverse engineering VI

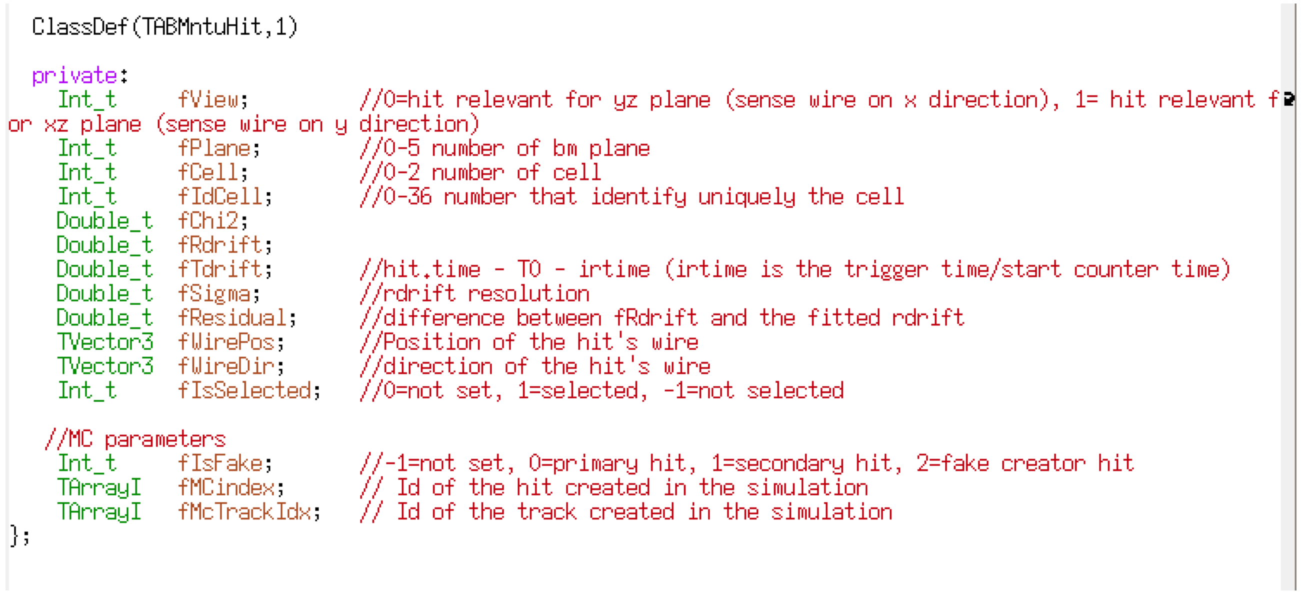
- Who is ‘saving’ the branch I am interested in and what it contains?
  - e.g.: the ‘bmrh’ branch…
- So.. inside shoe/libs/src..
  - grep bmrh TA*/* -> returns -> TABMbase/TABMntuRaw.cxx:TString TABMntuRaw::fgkBranchName = “bmrh.";
- Each ‘dataDsc’ fundamental object in shoe has a fgkBranchName.. Look for it in order to use it and understand how the object it is built!
- Usually:
  - you have several ‘ntuXXX’ classes. There’s a class that is holding the list of objects (e.g. in TABMntuRaw it holds the list of TAMBntuHit) and the methods to retrieve all the objects…
  - and there’s the class that describe each object: TABMntuHit.

A closer look inside BMntu

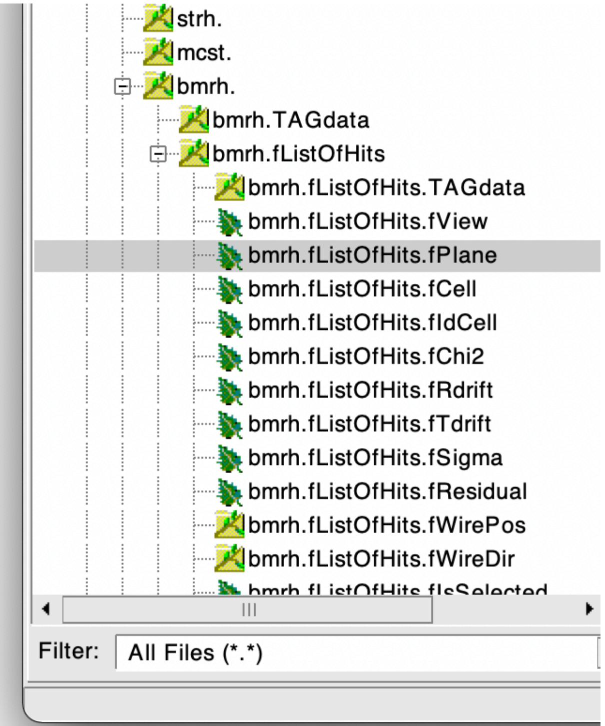
➡ TABMntuHit.hxx content…



```
class TABMntuHit : public TAGdata {
  public:
    TABMntuHit();
    TABMntuHit(Int_t iv, Int_t il, Int_t ic,Int_t id, Double_t r, Double_t t, Double_t s);
    virtual          ~TABMntuHit();
```



```
  ClassDef(TABMntuHit,1)

  private:
    Int_t     fView;        //0=hit relevant for yz plane (sense wire on x direction), 1= hit relevant f
or xz plane (sense wire on y direction)
    Int_t     fPlane;       //0-5 number of bm plane
    Int_t     fCell;        //0-2 number of cell
    Int_t     fIdCell;      //0-36 number that identify uniquely the cell
    Double_t  fChi2;
    Double_t  fRdrift;
    Double_t  fTdrift;      //hit.time - T0 - irtime (irtime is the trigger time/start counter time)
    Double_t  fSigma;       //rdrift resolution
    Double_t  fResidual;    //difference between fRdrift and the fitted rdrift
    TVector3  fWirePos;     //Position of the hit's wire
    TVector3  fWireDir;     //direction of the hit's wire
    Int_t     fIsSelected;  //0=not set, 1=selected, -1=not selected

   //MC parameters
    Int_t     fIsFake;      //-1=not set, 0=primary hit, 1=secondary hit, 2=fake creator hit
    TArrayI   fMCindex;     // Id of the hit created in the simulation
    TArrayI   fMcTrackIdx;  // Id of the track created in the simulation
};
```



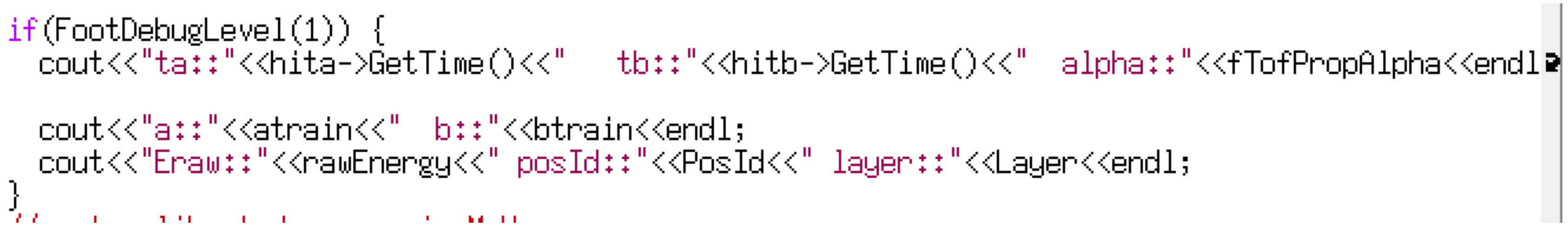
```
strh.
mcst.
bmrh.
  bmrh.TAGdata
  bmrh.fListOfHits
    bmrh.fListOfHits.TAGdata
    bmrh.fListOfHits.fView
    bmrh.fListOfHits.fPlane
    bmrh.fListOfHits.fCell
    bmrh.fListOfHits.fIdCell
    bmrh.fListOfHits.fChi2
    bmrh.fListOfHits.fRdrift
    bmrh.fListOfHits.fTdrift
    bmrh.fListOfHits.fSigma
    bmrh.fListOfHits.fResidual
    bmrh.fListOfHits.fWirePos
    bmrh.fListOfHits.fWireDir
    bmrh.fListOfHits.fIsSelected
Filter: All Files (*.*)
```

TAGdataDsc inside a tree

- So: ROOT can save TAGdataDsc objects directly inside a tree!
  - My action will 'build' the object I have defined, fill it with the relevant information and, on an event-by-event basis, store it inside the ntuple.
- The advantages are clear!
  - Eg: you can store 'a vertex' inside the tree and you can as to the given vertex which are the associated tracks! Not only double/ints or vectors.. but real 'shoe objects' can be stored.
  - Once you get your hands on a given pointer, you can use all the methods of the class to perform whatever action you want (eg. you can directly retrieve the tracks associated to a vertex using: TAVTvertex methods like GetTracksN, GetListOfTracks and GetTrack .. the last one returns a TAVTtrack* !)

Debugging

- How to add a debug info? A printout? How to control the messages from shoe?
  - To debug the code from time to time you'll need to add a 'printout' in your code. The level of verbosity can be checked using 'FootDebugLevel(XX)'..
  - XX is a number: if the 'debug' option passed inside FootGlobl.par is > than XX then the message will be printed. Eg:



```
if(FootDebugLevel(1)) {
  cout<<"ta::"<<hita->GetTime()<<"   tb::"<<hitb->GetTime()<<"  alpha::"<<fTofPropAlpha<<endl

  cout<<"a::"<<atrain<<"  b::"<<btrain<<endl;
  cout<<"Eraw::"<<rawEnergy<<" posId::"<<PosId<<" layer::"<<Layer<<endl;
}
```

is printed only if the debug level is >0. The higher the number, the larger is the verbosity of the output.. use it carefully!

- In order to better handle the debugging of different classes it is possibile to enable the 'debug' option only for 'one class' at a time.
  - See the talk from C. Finck tomorrow to 'tune' the output of each class!

Doing 'your' stuff..

- Now you should be able to contribute to the development of shoe. If you want to perform some kind of analysis or development of algorithms to be used in the shoe reconstruction (either level0 or fullrec)
  - Identify the package that provides the initial tools that you need
  - Identify the action that implements the algorithm you need, the input data and the output data as well as eventual geometrical/calibration and configuration information that you need.
  - Then you can code the action, and include it in the shoe executables using the BaseReco* implementation: you need to take care of understanding wether you need to end up inside the *MC* reco classes, the global reconstruction, the raw data reconstruction or else..
- But putting your hands inside shoe framework is not the only way to 'play' with the data and perform your analysis….

How to play with the output?

- Beside root interactive browser… Two main ways:
  - A ROOT **macro** or an **executable**
- For a quick access/check to the data, the macro solution is easier to implement and faster.. but if you need to perform complex analysis steps involving several classes, you might want to prepare a 'main' and compile it. [See tomorrow's talk from C. Finck]
- In the hands-on session we will be practicing macros.. just few forewords:
  - You can always try to look at the available macros under Reconstruction/level0 to get inspiration :)
  - the standard 'MakeClass()' approach on the root tree that contains shoe objects won't work: you need to play inside build/Reconstruction/level0 (to take advantage from the rootlogon.C macro) and remember to use the shoe objects and TAGroot framework to read the file.
- You can always have a look at TAGfoot/*Reco* classes to understand how to code the building blocks of a macro!
- **And remember: if the task gets too complicated.. best to go for executables that inherit from BaseReco!**

Troubleshooting

- Installation
  - Always look at the FOOT software twiki page.
  - Always give extended information about the step you went through, the architecture and computing environment (e.g. output of the g++ --version; uname -a; echo $ROOTSYS; root-config --cflags..)
- Runtime
  - that's a lot harder!!! :) there is plenty of things that can go wrong ..
  - Always give details about what input you are processing, which software you are using (macro, executable, etc), what are the config files that you are using (to provide this info you need to get your hands on the campaign manager.. not easy!) and what you are doing (data reco? MC reco? glb reco? from which campaign?.. etc etc)

Working with git

- The shoe software is in constant evolution!
- We use git to maintain the code. The baltig interface from INFN provides a web tool to 'navigate' the code and its changes. Otherwise you can use plain 'git' commands from command line to keep your code up to date.
- There's a wiki page documenting how to 'contribute' to shoe http://arpg-serv.ing2.uniroma1.it/twiki/bin/view/Main/FOOTDevelopers and play with git.
- Just three main recommendations:
  - keep your master branch up-to-date, keep your branch up-to-date with the master one. If the distance btw your branch and the master becomes too large, merging the algorithms will become a huge pain in the end!!
  - Observe the coding conventions as much as possible. This will ease the merging of your code within shoe!
  - Do not reinvent the wheel! we have tons of lines of code and examples. Just ask!