Playing with FOOT stuff

[an introduction…]

<u>A. S.</u>

with the kind help of S.M. Valle

Prerequisites

- ➡ You know about
  - C++
  - root
  - Fluka simulation
- ➡ You should have followed with GREAT ATTENTION the tutorial from Giuseppe yesterday
- ➡ Everything is supposed to work…. "requiring only minimal configuration and tweaking"….
  - at the current status of the software development we are basically around here….

The beginning…

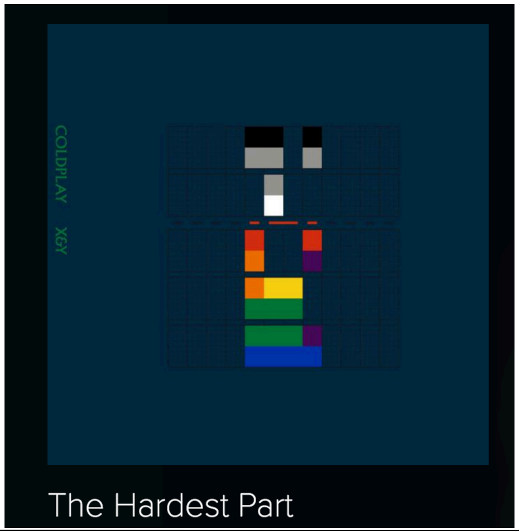
- Now that I have been taught on how to produce and tuple a root file…. what am I supposed to do?
  - Be happy! Producing the root file for the input was not an easy task anyway!
  - If someone produced the root file for you…. be happy anyway! Someone did the job for you!
- For the rest of the tutorial we assume that the input will be MC and that someone already did the job: tuples to be digested can be found on the cluster..
  - Root file: FOOT_EMFon*.root containing 500k events of 16O on C2H4 target (only events with inelastic interaction in the target where written on output, for compactness)
- The framework and its use are independent on the input. whenever something will be strictly dependent on the input, it will be stated.. Otherwise everything it is said is supposed to be working/running independently on data or MC.

Disclaimers

- Everything that is in place in inherited by FIRST: everything **HAS TO BE RECHECKED**
  - you cannot assume that it will work, and you have to re-understand how it is coded and why if you plan to use it.
- Most of the code is a "placeholder" for the future developments that will come. And it has to be re-implemented from scratch.
- Tips:
  - If you want to know what's available -> look at the header: .hxx
  - If you want to know how it's done -> look at the implementation: .cxx
- BEFORE coding new stuff check if something is already there!!!!
- For now, the only git command that you need to know (and operate with great care) is "git pull".
  - Before we can find ourselves discussing the terrific power of "git push" a guide for the developers will be prepared (by the software coordinator or whoever she/he will appoint for this task)

The hardest part

- Knowing what you have to do and where to put your hands.
- Strategy adopted here: start from use cases….
  1. I am a fan of Drift Chambers.. I'd like to play with them…. FOOT has a lot of them (2!). Where do I start?
  2. I am an addicted of TPC chambers readout with GEMs…. It seems that you are missing them in your FOOT "stuff".. Can I help?
  3. I want to know the truth. Can you show me the truth?
  4. I dream about Kalman Filters every night. Seems that you are still missing one.. I would be so happy to contribute….
- Those cases will be illustrated in detail, trying to provide guidance for the user needs that have been presented so far. This will be also the introductory part to the hands-on tutorial…



The DC fan (I)

- There's a detector that is already present in the framework/simulation and I have to work on it. Where do I find what I need?
  - The tools (libraries) needed to do what you have to do are under the framework folder:
    - TAG* folder: base classes (general purpose)
    - TAXX*base folders: implement what is specific for given detectors. XX decoding needs your imagination ;)
  - To understand what is available our DC friend can have a look inside:
    - TABMbase (Beam Monitor -> BM) coding the monitor before the target
    - TADCbase (Drift Chamber ->DC) coding the drift chamber after the magnets

ls -1 libs/src/
TAGbase
TAGfoot
TAGmclib

TABMbase
TACAbase
TADCbase
TAIRbase
TAITbase
TATWbase
TAVTbase

The DC fan (II): TABMbase

- There is indeed already a LOT of stuff!!!!
  - TABMact*: those are the actions executed by the framework at each call of NextEvent(). They take care of providing a given output (e.g. BM hits or BM tracks) starting from a given input (e.g. MC info or BM hits) and some external info (e.g. the BM geometry or cabling layout or calibration info)
  - TABMdat* and TABMntu* are the BM objects that are handling all the information that needs to be stored event by event: e.g. TABMdatRaw stores the "raw data", the TABMntuRaw stores the "collection of hits / hits info" the TABMntuTrack stores the "collection of tracks / track info".
  - TABMparGeo: interface to BM geometry
  - TABMparCon: interface to BM calibration
  - TABMparMap: interface to BM cabling

TABMactDatRaw.cxx
TABMactDatRaw.hxx
TABMactNtuMC.cxx
TABMactNtuMC.hxx
TABMactNtuRaw.cxx
TABMactNtuRaw.hxx
TABMactNtuTrack.cxx
TABMactNtuTrack.hxx
TABMdatRaw.LinkDef.h
TABMdatRaw.cxx
TABMdatRaw.hxx
TABMdatRaw.icc
TABMntuRaw.LinkDef.h
TABMntuRaw.cxx
TABMntuRaw.hxx
TABMntuRaw.icc
TABMntuTrack.LinkDef.h
TABMntuTrack.cxx
TABMntuTrack.hxx
TABMntuTrack.icc
TABMparCon.LinkDef.h
TABMparCon.cxx
TABMparCon.hxx
TABMparGeo.LinkDef.h
TABMparGeo.cxx
TABMparGeo.hxx
TABMparGeo.icc
TABMparMap.LinkDef.h
TABMparMap.cxx
TABMparMap.hxx
TABMvieTrackFIRST.cxx
TABMvieTrackFIRST.hxx

The DC fan (III): step 0

- I want to access what's already there.. (the plain MC info provided by fluka!)

```
new TABMactNtuMC("an_bmraw", myn_bmraw, myp_bmcon, myStr);
```

- I look for the action TABMactNtuMC that will
  - Take as input the MC truth (myStr)
  - Take as input the BM calibration (myp_bmcon)
  - Produce, as output, the tuples Hit information (myn_bmraw)
- To understand the input data: go to Giuseppe tutorial
- To understand the input BM calibration: go to the TABMparCon class
- To understand the output: go to the TABMntuRaw class
- **Beware:** the BM geometry is not used so far because in FIRST we had a "pre-processing" of the MC info that was producing a data-like MC output. **This is now missing and has to be implemented in the actNtuMC class explicitly calling the TABMparGeo class.**

The DC fan (IV): output

➡ The TABMntuRaw header: what I’m going to write in my tuple?

The TABMntuRaw class…
it’s just a collection of hits!

The TABMntuRawHit
class… implements the info!

```
class TABMntuRaw : public TAGdata {
  public:

                 TABMntuRaw();
   virtual       ~TABMntuRaw();

   TABMntuHit*      Hit(Int_t i_ind);
   const TABMntuHit* Hit(Int_t i_ind) const;

   virtual void    SetupClones();

   virtual void    Clear(Option_t* opt="");

   virtual void    ToStream(ostream& os=cout, Option_t* option="") const;

   ClassDef(TABMntuRaw,1)

  public:
   Int_t          nhit;              //
   TClonesArray*   h;                // hits
};
```

```
class TABMntuHit : public TObject {
  public:
    TABMntuHit();
    TABMntuHit(Int_t id, Int_t iv, Int_t il, Int_t ic, Double_t x,    Double_t y, Double_t z, Double_t px, Double_t …
…py, Double_t pz, Double_t r, Double_t t, Double_t tm);
    virtual        ~TABMntuHit();

    void           SetData(Int_t id, Int_t iv, Int_t il, Int_t ic, Double_t x,    Double_t y, Double_t z, Double_t px, D …
…ouble_t py, Double_t pz, Double_t r, Double_t t, Double_t tm);
    Int_t          Cell() const;
    Int_t          Plane() const;
    Int_t          View() const;
    Double_t X() const;
    Double_t Y() const;
    Double_t Z() const;
    TVector3 Position() const;
    Double_t Dist() const;
    Double_t Tdrift() const;
    Double_t Timmon() const;
    Bool_t HorView() const; //Horizontal, Top, XZ == -1
    Bool_t VertView() const; //Vertical, Side, YZ == 1

  private:
    Int_t idmon;
    Int_t iview;
    Int_t ilayer;
    Int_t icell;
    Int_t itrkass;
    Double_t ichi2;
    Double_t xcamon;
    Double_t ycamon;
    Double_t zcamon;
    Double_t pxcamon;
    Double_t pycamon;
    Double_t pzcamon;
    Double_t rdrift;
    Double_t tdrift;
    Double_t timmon;
    Double_t sigma;

    //Track related params
    TVector3  A0;
    TVector3  Wvers;

    Double_t  rho;
    TVector3  pca;
    TVector3  pca_wire;
```

The DC fan (V): the action

- Constructor declares input and output

```
TABMactNtuMC::TABMactNtuMC(const char* name,
                           TAGdataDsc* p_nturaw,
                           TAGparaDsc* p_parcon,
                           EVENT_STRUCT* evStr)
  : TAGaction(name, "TABMactNtuMC - NTuplize ToF raw data"),
    fpNtuMC(p_nturaw),
    fpParCon(p_parcon),
    fpEvtStr(evStr)
{
  Info("Action()"," Creating the Beam Monitor MC tuplizer action\n");
  AddPara(p_parcon, "TABMparCon");
  AddDataOut(p_nturaw, "TABMntuRaw");
}
```

- Action: loop on the MC hits and tuples the info

```
Bool_t TABMactNtuMC::Action()
{

  TAGgeoTrafo* fpFirstGeo = (TAGgeoTrafo*)gTAGroot->FindAction(TAGgeoTrafo::GetDefaultActName().Data());

  TABMntuRaw* p_nturaw = (TABMntuRaw*) fpNtuMC->Object();
  TABMparCon* p_parcon = (TABMparCon*) fpParCon->Object();


  Int_t nhits(0);
  if (!p_nturaw->h) p_nturaw->SetupClones();
  double locx, locy, locz;
  Double_t resolution;
  //The number of hits inside the BM is nmon
  Info("Action()","Processing n :: %2d hits \n",fpEvtStr->nmon);
  for (Int_t i = 0; i < fpEvtStr->nmon; i++) {

    /*
      Pre processing of INFO to compute the PCA info + rDrift and tDrift
    */
    /*
      write(*,*)'PCA= ',xca(ii), yca(ii), zca(ii)
      write(*,*)'p at PCA= ',pxca(ii), pyca(ii), pzca(ii)
      write(*,*)'rdrift= ',rdrift(ii),' tdrift= ', tdrift(ii),
    */

    //Tupling.
    if(i<32) {

      //X,Y and Z needs to be placed in Local coordinates.
      TVector3 gloc(fpEvtStr->xinmon[i],fpEvtStr->yinmon[i],fpEvtStr->zinmon[i]);
      TVector3 loc = fpFirstGeo->FromGlobalToBMLocal(gloc);
      locx = loc.X();
      locy = loc.Y();
      locz = loc.Z();

      //      resolution = p_parcon->ResoEval(fpEvtStr->rdrift[i]);
      resolution = p_parcon->ResoEval(0.1);
      //AS::: drift quantities have to be computed,
      TABMntuHit *mytmp = new((*(p_nturaw->h))[i])
        TABMntuHit(fpEvtStr->idmon[i],        fpEvtStr->iview[i],
                   fpEvtStr->ilayer[i],        fpEvtStr->icell[i],
                   locx, locy, locz, //Will become PCA
                   fpEvtStr->pxinmon[i],
                   fpEvtStr->pyinmon[i],        fpEvtStr->pzinmon[i], //will become mom @ PCA
                   0,          0, //here' rdrift tdrif.
                   fpEvtStr->timmon[i] );
      mytmp->SetSigma(resolution);
```

The DC fan (VI): possible ex.

- ➡ Implement access to truth link info!
  - – easy.
- ➡ Add the “real data like” reconstruction!
  - – hard. Requires to load in the action also the geometry, compute the PCA and drift info from fluka output….
- ➡ Implements / check the calibration!
  - – eeeasy.

```
Bool_t TABMactNtuMC::Action()
{

  TAGgeoTrafo* fpFirstGeo = (TAGgeoTrafo*)gTAGroot->FindAction(TAGgeoTrafo::GetDefaultActName().Data());

  TABMntuRaw* p_nturaw = (TABMntuRaw*) fpNtuMC->Object();
  TABMparCon* p_parcon = (TABMparCon*) fpParCon->Object();


  Int_t nhits(0);
  if (!p_nturaw->h) p_nturaw->SetupClones();
  double locx, locy, locz;
  Double_t resolution;
  //The number of hits inside the BM is nmon
  Info("Action()","Processing n :: %2d hits \n",fpEvtStr->nmon);
  for (Int_t i = 0; i < fpEvtStr->nmon; i++) {

    /*
      Pre processing of INFO to compute the PCA info + rDrift and tDrift
    */
    /*
      write(*,*)'PCA= ',xca(ii), yca(ii), zca(ii)
      write(*,*)'p at PCA= ',pxca(ii), pyca(ii), pzca(ii)
      write(*,*)'rdrift= ',rdrift(ii),' tdrift= ', tdrift(ii),
    */

    //Tupling.
    if(i<32) {

      //X,Y and Z needs to be placed in Local coordinates.
      TVector3 gloc(fpEvtStr->xinmon[i],fpEvtStr->yinmon[i],fpEvtStr->zinmon[i]);
      TVector3 loc = fpFirstGeo->FromGlobalToBMLocal(gloc);
      locx = loc.X();
      locy = loc.Y();
      locz = loc.Z();

      //      resolution = p_parcon->ResoEval(fpEvtStr->rdrift[i]);
      resolution = p_parcon->ResoEval(0.1);
      //AS::: drift quantities have to be computed,
      TABMntuHit* mytmp = new((*(p_nturaw->h))[i])
        TABMntuHit(fpEvtStr->idmon[i],          fpEvtStr->iview[i],
                   fpEvtStr->ilayer[i],         fpEvtStr->icell[i],
                   locx, locy, locz, //Will become PCA
                   fpEvtStr->pxinmon[i],
                   fpEvtStr->pyinmon[i],        fpEvtStr->pzinmon[i], //will become mom @ PCA
                   0,          0, //here' rdrift tdrif.
                   fpEvtStr->timmon[i] );
      mytmp->SetSigma(resolution);
```

Interlude: global geometry

- Each sub-detector has a local reference frame and lives inside a "box" that can be placed and rotated anywhere/in whichever way you want
- The transformations from the local and global FOOT frameworks are handled by the TAGgeoTrafo class. Such class is configured from a txt file present in the config/ folder in level0 project (currently there are DUMMY values that have to be checked / fixed / implemented).

```
Bool_t TABMactNtuMC::Action()
{
  TAGgeoTrafo* fpFirstGeo = (TAGgeoTrafo*)gTAGroot->FindAction(TAGgeoTrafo::GetDefaultActName().Data());

  TABMntuRaw* p_nturaw = (TABMntuRaw*) fpNtuMC->Object();
  TABMparCon* p_parcon = (TABMparCon*) fpParCon->Object();


  Int_t nhits(0);
  if (!p_nturaw->h) p_nturaw->SetupClones();
  double locx, locy, locz;
  Double_t resolution;
  //The number of hits inside the BM is nmon
  Info("Action()","Processing n :: %2d hits \n",fpEvtStr->nmon);
  for (Int_t i = 0; i < fpEvtStr->nmon; i++) {

    /*
      Pre processing of INFO to compute the PCA info + rDrift and tDrift
    */
    /*
      write(*,*)'PCA= ',xca(ii), yca(ii), zca(ii)
      write(*,*)'p at PCA= ',pxca(ii), pyca(ii), pzca(ii)
      write(*,*)'rdrift= ',rdrift(ii),' tdrift= ', tdrift(ii),
    */

    //Tupling.
    if(i<32) {

      //X,Y and Z needs to be placed in Local coordinates.
      TVector3 gloc(fpEvtStr->xinmon[i],fpEvtStr->yinmon[i],fpEvtStr->zinmon[i]);
      TVector3 loc = fpFirstGeo->FromGlobalToBMLocal(gloc);
      locx = loc.X();
      locy = loc.Y();
      locz = loc.Z();
```

The TPC+ GEM addicted

- In this case, I have really a few things to offer…. (as TPC+GEM where never included in the framework)
  - So it is a **perfect example** for anyone that wants to code something from scratch!
- How to include the TPC+GEM inside our decoding code?
  1. Build the TATGbase empty folder, prepare a Makefile (take it from the TAIT or TAIR folders and change the file names accordingly)
  2. Define the data structure: what information will the MC produce that you have to "tuple"? Hits? Tracks? Then you have to prepare a TATGntuXxx class containing what you need
  3. Define the MC tupling action: write a class that takes as input the MC info and produce the TAGntuXxx object (see what is already here for the BM or other existing detector).
  4. Define the calibration and geometry classes for the detector

The truth seeker (I)

- As we still live in the wonderful world of MC.. at some point it would be good to access… “THE TRUTH”.
- There’s a class that is already available and does the job for you:
  - TAGntuMCeve is the TAGdata that contains the track/particle block (see giuseppe’s talk from yesterday)
  - TAGactNtuMC: takes the track block from root file and dumps it into the TAGdata

```
class TAGntuMCeve : public TAGdata {
  public:
                    TAGntuMCeve();
    virtual          ~TAGntuMCeve();

    TAGntuMCeveHit*       Hit(Int_t i);
    const TAGntuMCeveHit* Hit(Int_t i) const;

    virtual void    SetupClones();
    virtual void    Clear(Option_t* opt="");

    virtual void    ToStream(ostream& os=cout, Option_t* option="") const;

    ClassDef(TAGntuMCeve,1)

  public:
    Short_t         nhit;               // nhit
    TClonesArray*    h;                   // hits
};

#include "TAGntuMCeve.icc"
```

```
class TAGntuMCeveHit : public TObject {
  public:
                  TAGntuMCeveHit();

  // VM changed 17/11/13 to add information for simulate pile-up events
  TAGntuMCeveHit(Int_t i_id, Int_t i_chg, Int_t i_type,
                 Int_t i_reg, Int_t i_bar, Int_t i_dead,
                 Double_t i_mass, Int_t i_moth,
                 Double_t i_time,
                 Double_t i_tof, Double_t i_trlen,
                 TVector3 i_ipos, TVector3 i_fpos,
                 TVector3 i_ip,TVector3 i_fp,
                 TVector3 i_mothip,
                 TVector3 i_mothfp,Int_t i_pileup);

  virtual        ~TAGntuMCeveHit();
```

```
Int_t       id;          // fluka identity
Int_t       chg;         // charge
Int_t       type;        // Type
Int_t       reg;         // region
Int_t       bar;         // barionic number
Int_t       dead;        // region in whic die
Double_t    mass;        // mass
Int_t       mothid;      // mother identity
Double_t    time;        // time of produ
Double_t    tof;         // time of flight
Double_t    trlen;       // track lenght
TVector3    inpos;       // initial position
TVector3    fipos;       // final position
TVector3    ip;          // initial momentum
TVector3    fp;          // final momentum
TVector3    mothip;      // mother init momentum
TVector3    mothfp;      // mother final momentum
Int_t       pileup;      // pile-up index =0 current event
                         // >0 index of overlapped event
                         // VM added 17/11/13
```

The truth seeker (II)

- In my "level0" job I can then define the truth tupling action, define the MCeve data output and add it to the output ntuple…
- After calling "NextEvent()" I have then access at the track block for each event and I can use it and navigate it as Giuseppe explained in the simulation tutorial
  - I can retrieve the TAGntuMCeve class and access the hits info trough the TAGntuMCeveHit methods…

```
void RecoTools::FillMCEvent(EVENT_STRUCT *myStr) {

  /*Ntupling the general MC event information*/
  myn_mceve    = new TAGdataDsc("myn_mceve", new TAGntuMCeve());

  new TAGactNtuMCeve("an_mceve", myn_mceve, myStr);

  my_out->SetupElementBranch(myn_mceve,     "mceve.");
```

```
    tagr->NextEvent();

    //do some MC check
    //to be moved to framework
    if(m_doVertex)
      AssociateHitsToParticle();
```

```
void RecoTools::AssociateHitsToParticle() {


  TAGntuMCeve*  p_ntumceve =
    (TAGntuMCeve*)   myn_mceve->GenerateObject();

  vector<int> FragIdxs;
  int nhitmc = p_ntumceve->nhit;
  for(int i=0; i<nhitmc; i++){
    TAGntuMCeveHit *myPart = p_ntumceve->Hit(i);

    int part_reg = myPart->Reg();
```

The truth seeker (III)

- But… what if I am interested in what happened in a given detector and I want to relate what happened to the particles that have interacted with my detector?
  - It's already possible for most of the detectors (not all of them have already implemented the necessary changes in the data class) trough the machinery explained by Giuseppe in yesterday's talk (pointer of id-1 to the track block!)
  - Exercise from Giuseppe can be easily re-implemented in the framework!

```
TAGntuMCeve*  p_ntumceve =
  (TAGntuMCeve*)  myn_mceve->GenerateObject();

int nhitmc = p_ntumceve->nhit;
 for(int i=0; i<nhitmc; i++){
  TAGntuMCeveHit *myPart = p_ntumceve->Hit(i);

  int part_reg = myPart->Reg();

  //Require that particle is produced inside the TGT
  if(part_reg == 3) {
    FragIdxs.push_back(i);
  }
 }
```

Retrieve the info of MC eve from the ROOT memory

```
 //Pixels stuff
 TAVTntuRaw*  p_nturaw =
  (TAVTntuRaw*)  myn_vtraw->GenerateObject();

 for(int t_frg = 0; t_frg<FragIdxs.size(); t_frg++) {

  //Check VTX pixels
  for(int i=0; i<p_nturaw->GetPixelsN(0); i++){
   hit = p_nturaw->GetPixel(0,i);
   tmp_vtxid = hit->GetMCid()-1;
   if(tmp_vtxid == FragIdxs.at(t_frg)){
    if(m_debug) cout<<" Vtx hit associated to part "<<t_frg<<" That is a:: "<<p_ntumceve->Hit(t_frg)->Fluk
aID()<<"and has charge, mass:: "<<p_ntumceve->Hit(t_frg)->Chg()<<" "<<p_ntumceve->Hit(t_frg)->Mass()<
<" "<<endl;
   }
  }
```

Associate the TAVT hits to the particle in the track block using the tmp_vtxid

The Kalman Dreamer (I)

- You have problems… big ones.. However, let's provide some help…
- This is "high level" stuff that will end up Reconstruction/fullrec project that is currently empty..
- It is high level since… it requires that the trackers have already done their jobs reconstructing hits applying geometry and calibration info.. relative position of detectors is also needed…
  - An executable will be prepared in the near future that will take as input the output of level0 and provide all the info needed for high level actions…
- Does this means that nothing can be done for now?
  - NOT AT ALL :D !!!!
- Let's try to see what can be done "right now"…..

The Kalman Dreamer (II)

- What do I need as input?
  - Hits from the trackers / detectors that belong to a given track/event.
- How can I access this info?
  - Not that hard: as soon as I call "NextEvent()" all the level0 actions are executed and I have at my disposal, in the ROOT memory, what I need. This means that in RecoTools.cc, event by event, once I have called/decoded the trackers I can retrieve the info and use it in my "custom Kalman" code.
- The near future: code an action that uses the info of VT (vertex detector), IT (Inner tracker) and DC (Drift chamber) and fill a vector of hits associated to a given "true particle".

```
for (Long64_t jentry=0; jentry<nentries;jentry++) {
  if(m_debug) cout<<" New Eve "<<endl;
  nb = tree->GetEntry(jentry);   nbytes += nb;
  // if (Cut(ientry) < 0) continue;

  tagr->NextEvent();


   //Pixels stuff
   TAVTntuRaw*  p_nturaw =
     (TAVTntuRaw*)   myn_vtraw->GenerateObject();


  TAGntuMCeve*  p_ntumceve =
    (TAGntuMCeve*)   myn_mceve->GenerateObject();
```

The Kalman Dreamer (III)

- How it should be done
  - Implement the "event reconstruction" in TAGfoot using the framework: e.g. TAGactGlbTracking in TAGfoot
- The action (TAGactGlbTrackingMC) will need to load the VT, IT, DC classes (and maybe also the TW and CA for forward extrapolation)
  - Then will have to implement a "forward tracking" method that loads the hits and uses them to build a tracks with Kalman filtering turned on
  - Examples can be found in GlobalTrack class.

```
TAGactGlbTracking::TAGactGlbTracking(const char* name, TAGdataDsc* p_glb,
                                     TAGdataDsc* p_vtvtx, TAGdataDsc* p_traw,
                                     TAGparaDsc* p_geotof, double Current,
                                     TString dir, TAGdataDsc* p_mceve) :
  TAGaction(name, "TAGactGlbTracking - Global Tracking"),
  fpGlb(p_glb),
  fpVtvtx(p_vtvtx), fpTraw(p_traw),
  fpParTofGeo(p_geotof)
{

  AddDataOut(p_glb, "TAGntuGlbTracks");
  AddDataIn(p_vtvtx, "TAVTntuVertex");
  AddDataIn(p_traw, "TATntuRaw");
  fpMCeve = p_mceve;
  if(p_mceve)
    AddDataIn(p_mceve, "TAGntuMCeve");

  // changed from 700. to 730 (to be fined tuned - VM)
  Double_t AladinCurrent = 0.;  //Default is without magfield
  // Set in HlRecoTools via SetCurrent
  if(Current)
    AladinCurrent = Current;

  fField    = new MagneticField(AladinCurrent,dir);
  fGloTrack = new GlobalTrack(fField);//TODO: stuff geo info in here
```

FIRST example

```
fGloTrack->ClearTracks();
//Execute tracking algorithm only if there are at least a vtx track and 1 hit on the tof

if(on_vtvtx->GetVertexN() && on_traw->nhit)
  fGloTrack->MakeGlobalTracksForward(on_vtvtx, on_traw, p_tofgeo, p_mcblk);
```

The Kalman Dreamer (IV)

- How can you do it if you are in a hurry
  - Inside RecoTools, after calling NextEvent(), run the hit association routine, get a vector of points from each detector and pass it to your favourite Kalman tool.
  - Instead of the "debug info" add the hits inside a vector and feed them to the Kalman tool

```
int nhitmc = p_ntumceve->nhit;
 for(int i=0; i<nhitmc; i++){
   TAGntuMCeveHit *myPart = p_ntumceve->Hit(i);

   int part_reg = myPart->Reg();

   //Require that particle is produced inside the TGT
   if(part_reg == 3) {
     FragIdxs.push_back(i);
   }
 }

 //Pixels stuff
 TAVTntuRaw*  p_nturaw =
   (TAVTntuRaw*)  myn_vtraw->GenerateObject();

 for(int t_frg = 0; t_frg<FragIdxs.size(); t_frg++) {

   //Check VTX pixels
   for(int i=0; i<p_nturaw->GetPixelsN(0); i++){
     hit = p_nturaw->GetPixel(0,i);
     tmp_vtxid = hit->GetMCid()-1;
     if(tmp_vtxid == FragIdxs.at(t_frg)){
       if(m_debug) cout<<" Vtx hit associated to part "<<t_frg<<" That is a:: "<<p_ntumceve->Hit(t_frg)->Fluk
aID()<<"and has charge, mass:: "<<p_ntumceve->Hit(t_frg)->Chg()<<" "<<p_ntumceve->Hit(t_frg)->Mass()<
<" "<<endl;
     }
   }

   //Check IT pixels
   for(int i=0; i<p_itnturaw->GetPixelsN(0); i++){
     hitIT = p_itnturaw->GetPixel(0,i);
     tmp_itid = hitIT->GetMCid()-1;
     if(tmp_itid == FragIdxs.at(t_frg)){
       if(m_debug) cout<<" IT hit associated to part "<<t_frg<<" That is a:: "<<p_ntumceve->Hit(t_frg)->Fluka
ID()<<"and has charge, mass:: "<<p_ntumceve->Hit(t_frg)->Chg()<<" "<<p_ntumceve->Hit(t_frg)->Mass()<
<" "<<endl;
     }
   }
```

The Kalman Dreamer (V)

- The Kalman code:
  - http://genfit.sourceforge.net/Main.html can be (on request) easily implemented.
  - Any other custom solution is welcomed.

2nd interlude: Mag field

- The "Sanelli magnets" map is not yet available for now….
- There's an interface class already available:
  - MagneticField.* inside TAGfoot. This class loads/provide the FIRST magnetic field. **Still has to be updated in order to handle the FOOT mag field!**
  - **Volunteers? :D**

The real deal..

- Beside playing with exercises… As for the "general framework" we have a lot of real work to do [not in relevance order]:
  1. Provide a transparent interface of FOOT geometry setup to the users. Classes are there, we need to define the detectors RF positions and code what is needed.
  2. Provide an interface to the magnetic field
  3. Update the (3D) event display (this can happen ONLY after 1 is accomplished)
- Developers
  - While it is true that now we're playing with MC.. bear in mind that at some point we're going to have real data for input! So: design the data classes in order to be as much transparent as possibile in the migration from data to MC

Conclusions

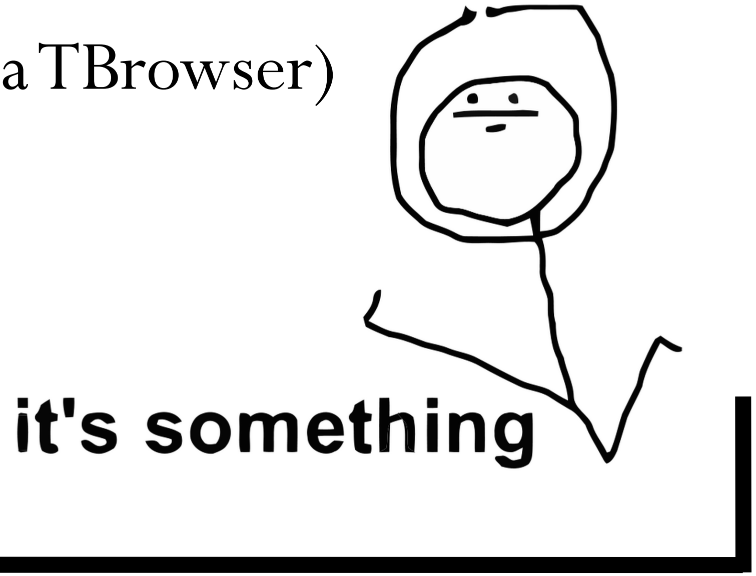
- Problems with the framework? See the talk at the previous meeting.
- Problems with git? Enjoy this tutorial
- When adding new stuff to the output, please check carefully your code for memory leaks.
- Before "pushing" to the repository a strategy for the software management has to be defined and a responsible for the software needs to be appointed (check for possible conflicts, release only bullet proof code, coordinate activities on level0 and fullrec projects, etc etc)
- **Please document the work you are doing in the Twiki page: http://arpg-serv.ing2.uniroma1.it/twiki/bin/view/Main/FOOTSoftware** other pages and links can be added accordingly to the user will….

and now… let's play

Exerciseeeeeesssss….. I'm coooooming…..

Ex 0: the newcomer

- Go to http://arpg-serv.ing2.uniroma1.it/twiki/bin/view/Main/FOOTSoftware and follow the instruction to:
  - Get the code from git.
  - Compile the framework.
  - Compile DecodeMC in level0
- Run the analizer
  - ./DecodeMC -in /home/FOOT-T3/battistfoott3/SoftDemo/FOOT_EMFon.root -out MyDCfanExe.root
- Check the output (navigate the root file using a TBrowser)
- ….. be happy!



Ex 1: the DC fan

- Go to http://arpg-serv.ing2.uniroma1.it/twiki/bin/view/Main/FOOTSoftware and follow the instruction to:
  - Get the code from git.
  - Compile the framework.
  - Compile DecodeMC in level0
- Find inside RecoTools.cc the call to TABMactNtuMC and check if the tupling of the BM info is "turned on".
- Run the analizer
  - ./DecodeMC -in /home/FOOT-T3/battistfoott3/SoftDemo/FOOT_EMFon.root -out MyDCfanExe.root
- Then open the framework action in libs/src/TABMbase/TABMactNtuMC.cxx , do your exe [Slide 11], recompile the framework, rerun the analizer and….. be happy!

Ex 2: adding new detector..

- .. no … you don't want to do that…
- But, maybe, you want to put your hands on the "latest" added detectors:
  - Inner tracker: inherits code from VTX, but only two planes. To be checked thoroughly….
  - Drift chamber: inherits code from BM, to be checked
  - TW: dummy. To be coded ~ from scratch.. inherits from SC… missing both calibration and geometry helper classes… only truth info is coded for now.
  - CA: dummy as TW.
- So, if you really want to play with "new stuff":
  - Go in the TATWbase folder, add the info you'd like to browse (eg. momentum of the particle interacting with scintillator or with calo) to the TAxxntuRaw class, and then add the tupling to the action TAxxactNtuMC
  - Then go into l0reco, rerun the analizer and ….. be happy!

Ex 3: the truth seeker

- Go to the Giuseppe's slides: https://agenda.infn.it/conferenceDisplay.py?confId=12219
  - Take the course and go to exercise A (slide 31 https://agenda.infn.it/getFile.py/access?contribId=0&resId=4&materialId=slides&confId=12219)
  - Code the exercise inside RecoTools.cc
  - rerun the analizer and….. be happy!
- Find the void RecoTools::AssociateHitsToParticle() call inside RecoTools..
  - Add the hits of DC and TW (tof wall, or scintillator) to the method and retrieve the particles that in a given event are firing ALL the 4 detectors (VT, IT, DC and TW).
  - Recompile RecoTools.cc
  - rerun the analizer and….. be happy!

Ex 4: the Kalman dreamer

- Go to slides 19, 20.
- Choose your preferred approach to build a list of hits associated to a given particle.
- Feed the list to whatever algorithm you have/want to test..... and..... be happy!
- Of course the last one is the hard part of the game. We do not provide yet an interface to a Kalman filtering code. But we can provide anything you want : just ask it…
  - Then, of course, the Kalman code will require as input the magnet field and the geometry… We need to sit down and understand the best way to provide this info within the framework….