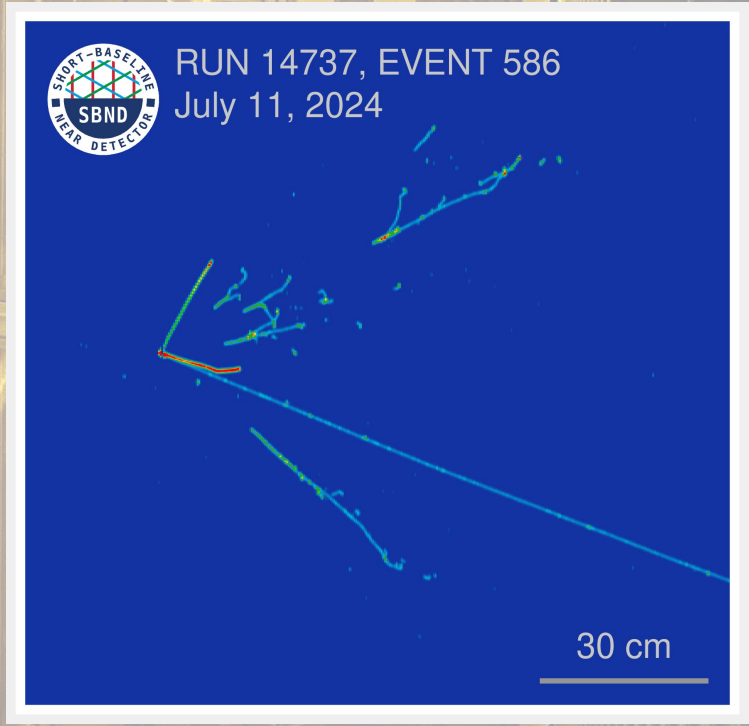
Writing your First Analyz(s)er

28th October 2026

11th UK LArTPC Software & Analysis Workshop

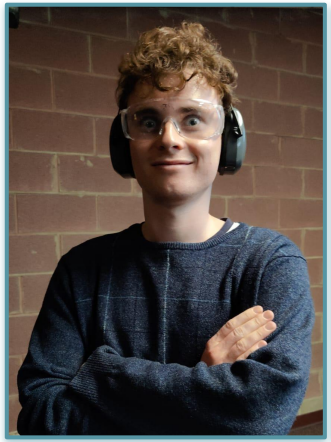
Isobel Mawby & Anna Beever/Dom Brailsford
i.mawby1@lancaster.ac.uk &
albeever1@sheffield.ac.uk
#analysis

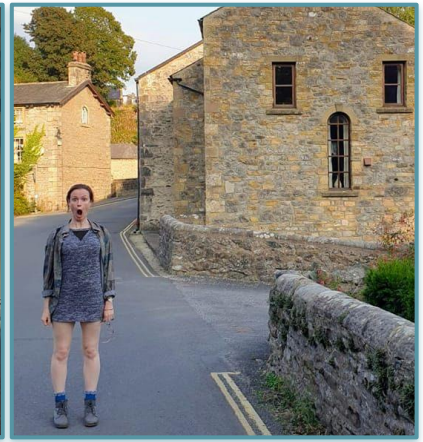


Overview & aims of this session

- Learn how to do some physics with the reconstructed events you produced
  - Don't worry if you didn't manage to make the files, I'll point you to some we've made
- Learn how to access the reconstructed neutrino information
  - There is a generic procedure for accessing almost all of the neutrino information you have in every file you've made this week
- We'll look at:
  - Reconstruction objects produced by Pandora and downstream reconstruction
  - Associations of these objects to higher-level information
  - Take your time & try to understand everything you do
- Hopefully we'll be able to make some plots

Thanks to all who have given this tutorial over the last few years, these slides have been (very marginally) adapted from those previous versions.












Side note

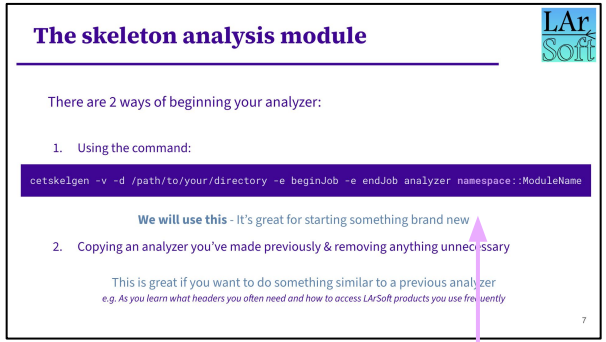
- We have included what will probably be far too much to achieve in these sessions
- But hopefully it's all structured clearly enough that you can continue with the exercises in your own time
- So please don't worry if you don't make it hugely far through this tutorial, there's supposed to be too much content
- If you are reading these slides as a PDF, you might prefer to look at the Google Slides link explicitly, as some code blocks render better there

Slide Structure

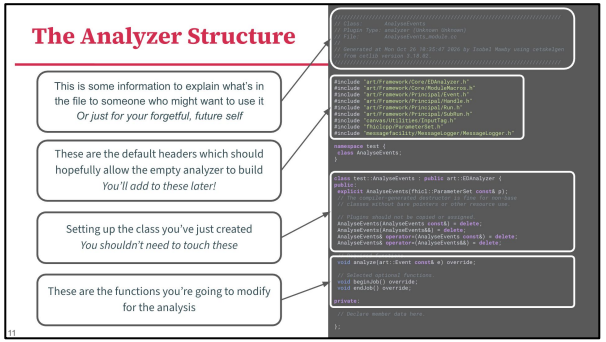
‘New Topic’ Slide



‘Lecture’ Slide



‘Exercise’ Slide



The pink text indicates places where you need to replace the line with your personal version



1. The **Analyzer** Skeleton

The skeleton analysis module

There are 2 ways of beginning your analyzer:

1. Using the command:

```
cetskelgen -v -d /path/to/your/directory -e beginJob -e endJob analyzer namespace::ModuleName
```

We will use this - It's great for starting something brand new

2. Copying an analyzer you've made previously & removing anything unnecessary

This is great if you want to do something similar to a previous analyzer

e.g. As you learn what headers you often need and how to access LArSoft products you use frequently

cetskelgen

These are optional functions which will be added to your analyzer, we'll look at them in the next few slides

```
cetskelgen -v -d /path/to/your/directory -e beginJob -e endJob analyzer namespace::ModuleName
```

Choose something sensible here, e.g. **`test::AnalyseEvents`**

For more information, see:
`https://cdcvs.fnal.gov/redmine/projects/cetlib/wiki/Cetskelgen`

Let's do it!

Have you remembered to setup the SL7 container?

If you are using a fresh terminal you will need to setup again:

```
source /cvmfs/sbnd.opensciencegrid.org/products/sbnd/setup_sbnd.sh
source /PATH/TO/YOUR/BUILD/AREA/localProducts*/setup
mrbslp
```

1. Navigate here:

We've put the `CMakeLists.txt` and `build.sh` files here...

```
cd $MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis
```

2. Type the cetskelgen command:

The full stop tells `cetskelgen` to place the analysis module in the current directory

```
cetskelgen -v -d . -e beginJob -e endJob analyzer test::AnalyseEvents
```

What did we create?

- You should now find a file called `AnalyseEvents_module.cc`, this is your analyzer!
- Open this!
- The top section should look something like the snippet on the right

```
#include "art/Framework/Core/EDAnalyzer.h"
#include "art/Framework/Core/ModuleMacros.h"
#include "art/Framework/Principal/Event.h"
#include "art/Framework/Principal/Handle.h"
#include "art/Framework/Principal/Run.h"
#include "art/Framework/Principal/SubRun.h"
#include "canvas/Utilities/InputTag.h"
#include "fhiclcpp/ParameterSet.h"
#include "messagefacility/MessageLogger/MessageLogger.h"

namespace test {
 class AnalyseEvents;
}


class test::AnalyseEvents : public art::EDAnalyzer {
public:
 explicit AnalyseEvents(fhicl::ParameterSet const& p);
 // The compiler-generated destructor is fine for non-base
 // classes without bare pointers or other resource use.

 // Plugins should not be copied or assigned.
 AnalyseEvents(AnalyseEvents const&) = delete;
 AnalyseEvents(AnalyseEvents&&) = delete;
 AnalyseEvents& operator=(AnalyseEvents const&) = delete;
 AnalyseEvents& operator=(AnalyseEvents&&) = delete;

 // Required functions.
 void analyze(art::Event const& e) override;

 // Selected optional functions.
 void beginJob() override;
 void endJob() override;

private:

 // Declare member data here.

};


test::AnalyseEvents::AnalyseEvents(fhicl::ParameterSet const& p)
 : EDAnalyzer{p}  // ,
 // More initializers here.
{
 // Call appropriate consumes<>() for any products to be retrieved by this module.
}

void test::AnalyseEvents::analyze(art::Event const& e)
{
 // Implementation of required member function here.
}

void test::AnalyseEvents::beginJob()
{
 // Implementation of optional member function here.
}

void test::AnalyseEvents::endJob()
{
 // Implementation of optional member function here.
}

DEFINE_ART_MODULE(test::AnalyseEvents)
```

The Analyzer Structure

This is some information to explain what's in the file to someone who might want to use it
Or just for your forgetful, future self

These are the default headers which should hopefully allow the empty analyzer to build
You'll add to these later!

Setting up the class you've just created
You shouldn't need to touch these

These are the functions you're going to modify for the analysis

```
////////////////////////////////////////////////////////////////////////
// Class:       AnalyseEvents
// Plugin Type: analyzer (Unknown Unknown)
// File:        AnalyseEvents_module.cc
//
// Generated at Mon Oct 26 10:35:47 2026 by Isobel Mawby using cetskelgen
// from cetlib version 3.18.02.
////////////////////////////////////////////////////////////////////////
```

```
#include "art/Framework/Core/EDAnalyzer.h"
#include "art/Framework/Core/ModuleMacros.h"
#include "art/Framework/Principal/Event.h"
#include "art/Framework/Principal/Handle.h"
#include "art/Framework/Principal/Run.h"
#include "art/Framework/Principal/SubRun.h"
#include "canvas/Utilities/InputTag.h"
#include "fhiclcpp/ParameterSet.h"
#include "messagefacility/MessageLogger/MessageLogger.h"

namespace test {
 class AnalyseEvents;
}
```

```
class test::AnalyseEvents : public art::EDAnalyzer {
public:
 explicit AnalyseEvents(fhicl::ParameterSet const& p);
 // The compiler-generated destructor is fine for non-base
 // classes without bare pointers or other resource use.

 // Plugins should not be copied or assigned.
 AnalyseEvents(AnalyseEvents const&) = delete;
 AnalyseEvents(AnalyseEvents&&) = delete;
 AnalyseEvents& operator=(AnalyseEvents const&) = delete;
 AnalyseEvents& operator=(AnalyseEvents&&) = delete;
```

```
 // Required functions.
 void analyze(art::Event const& e) override;

 // Selected optional functions.
 void beginJob() override;
 void endJob() override;

private:

 // Declare member data here.

};
```

The Analyzer Structure

This is the constructor, we'll access configuration parameters here later on

```
test::AnalyseEvents::AnalyseEvents(fhicl::ParameterSet
const& p)
 : EDAnalyzer{p}  // ,
 // More initializers here.
{
 // Call appropriate consumes<>() for any products to be
retrieved by this module.
}
```

This is the analyze function, it's called for every event you give it in the LArSoft job

```
void test::AnalyseEvents::analyze(art::Event const& e)
{
 // Implementation of required member function here.
}
```

These optional functions are called once, before and after any and all events are analyzed

```
void test::AnalyseEvents::beginJob()
{
 // Implementation of optional member function here.
}


void test::AnalyseEvents::endJob()
{
 // Implementation of optional member function here.
}
```

Macro to tell art that this module exists
This is used in the fcl configuration in a few slides

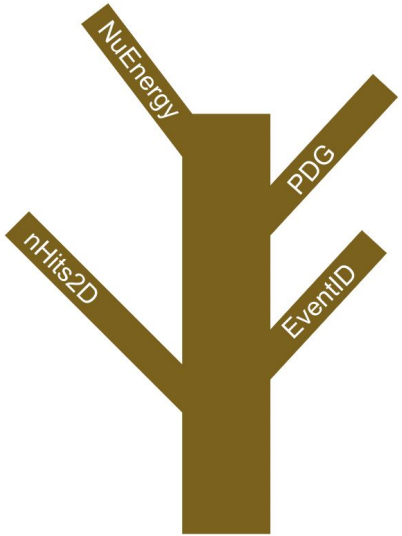
```
DEFINE_ART_MODULE(test::AnalyseEvents)
```

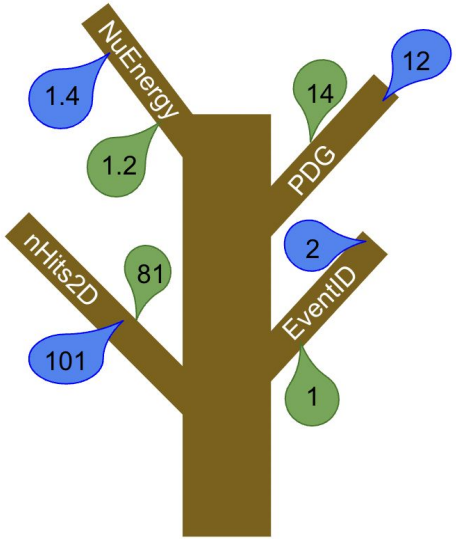


2. Obtaining Our First Analysis Information

Writing out Analysis Information

1) We're going to create a ROOT TTree to store our analysis information







2) Then we will add to our tree, the 'Event ID' of our created events

https://root.cern.ch/doc/master/classTTree.html

Creating a TTree

Add relevant LArSoft & ROOT headers

```
// Additional framework includes
#include "art_root_io/TFileService.h"
// ROOT includes
#include <TTree.h>
```

Declare TTree

```
private:
 // Create output TTree
 TTree *fTree;
```

Create your TTree

```
void test::AnalyseEvents::beginJob()
{
 // Get the TFileService to create the output TTree for us
 art::ServiceHandle<art::TFileService> tfs;
 fTree = tfs->make<TTree>("tree", "Output TTree");
}
```

Note: The order represents their locations in the file

Writing Out a Variable

Declare event-based variables

Access our event ID from the LArSoft event we're analysing & fill the TTree

Add branches for the variables we want to fill

```
private:
 // Create output TTree
 TTree *fTree;

 // Tree variables
 unsigned int fEventID;
```

```
void test::AnalyseEvents::analyze(art::Event const& e)
{
 // Set the event ID
 fEventID = e.id().event();

 // Fill tree
 fTree->Fill();
}
```

```
void test::AnalyseEvents::beginJob()
{
 // Get the TFileService to create the output TTree for us
 art::ServiceHandle<art::TFileService> tfs;
 fTree = tfs->make<TTree>("tree", "Output TTree");

 // Add branches to TTree
 fTree->Branch("eventID", &fEventID);
}
```

Running the analysis module

In order to be able to run the analyzer, we now need to write 2 fhicl files

- The first will configure our analyser - **an include fcl**
  - This is where we point the analyzer to the objects/parameters we want to access from the input files (this will make more sense soon…)
- The second will be used to run our analyser **- a run/job fcl**
  - This links together the configuration file and the analysis module
- The main reason we don't just define our parameters in the run/job fcl is that multiple run/job fcls can all inherit from the include fcl. This way we reduce our points of maintenance.

Fhicl 1: Configuring the analyzer

Fhicl 1: Configuring the analyzer. Create a file, e.g. **`analysisConfig.fcl`** & fill it with this:



Later this is where we will add any configuration of our analyzer module.

Fhicl 2: Running the module

Include your analyzer configuration fhicl

Name this process

Must not include any underscores

Tell it to expect a ROOT input file

Output filename

ana sets our module **analyzeEvents** as part of the workflow

Note, this matches the name in the configuration fcl file

```
#include "analysisConfig.fcl"
#include "simulationservices_sbnd.fcl"

process_name: AnalyseEvents  # The process name must NOT
contain any underscores

source:
{
  module_type: RootInput  # Telling art we want a ROOT input
  maxEvents:   -1
}

services:
{
  TFileService: { fileName: "analysisOutput.root" }
  @table::sbnd_services
}

physics:
{
  analyzers:
  {
    ana: @local::analyseEvents
  }

  path0:     [ ana ]
  end_paths: [ path0 ]
}
```



Let’s try running it…

Pre-made reconstructed events

Haven’t made a reconstruction file? Don’t panic!

There is a pre-made reconstruction file which can be found here:

`/Workshop/sim_testsamples/reco2_withmuon_10evts.root`

Compiling and running your code

First, we need to compile what you've written so far

From the `$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis` directory:

```
source build.sh
```

This has each build command in one place, have a look to make sure you're comfortable with what it does before using it

Then (when successful) run your analyzer!

```
lar -c run_analyseEvents.fcl -s /path/to/input/file.root -n 10
```

Let's just run over 10 events while we make sure things build.
We'll run on the whole sample later

Open the file in ROOT to investigate our output file...

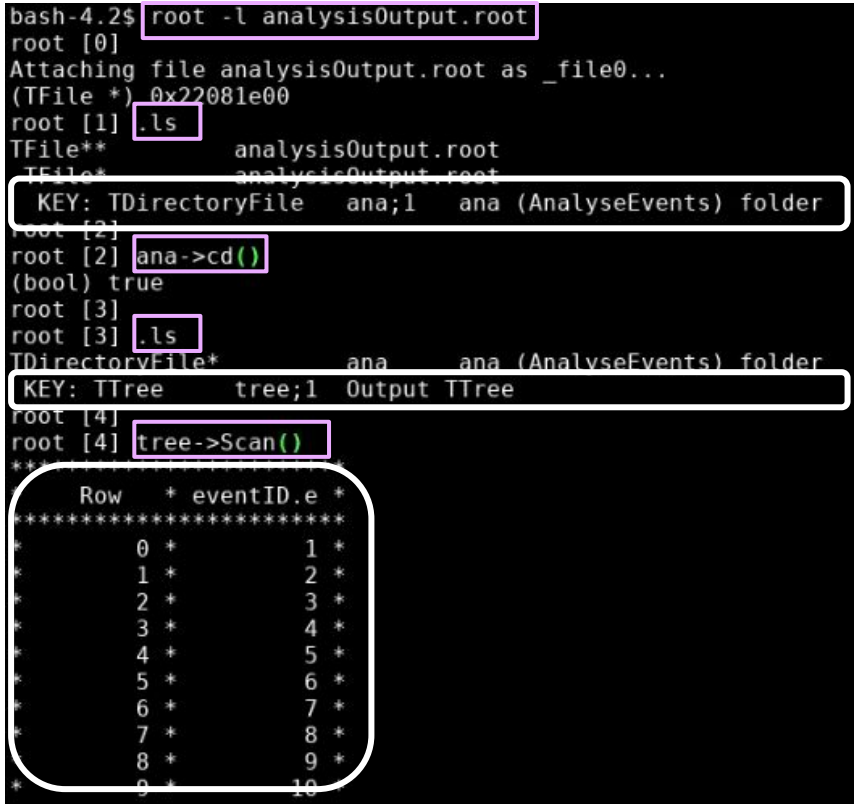
```
root -l analysisOutput.root
```

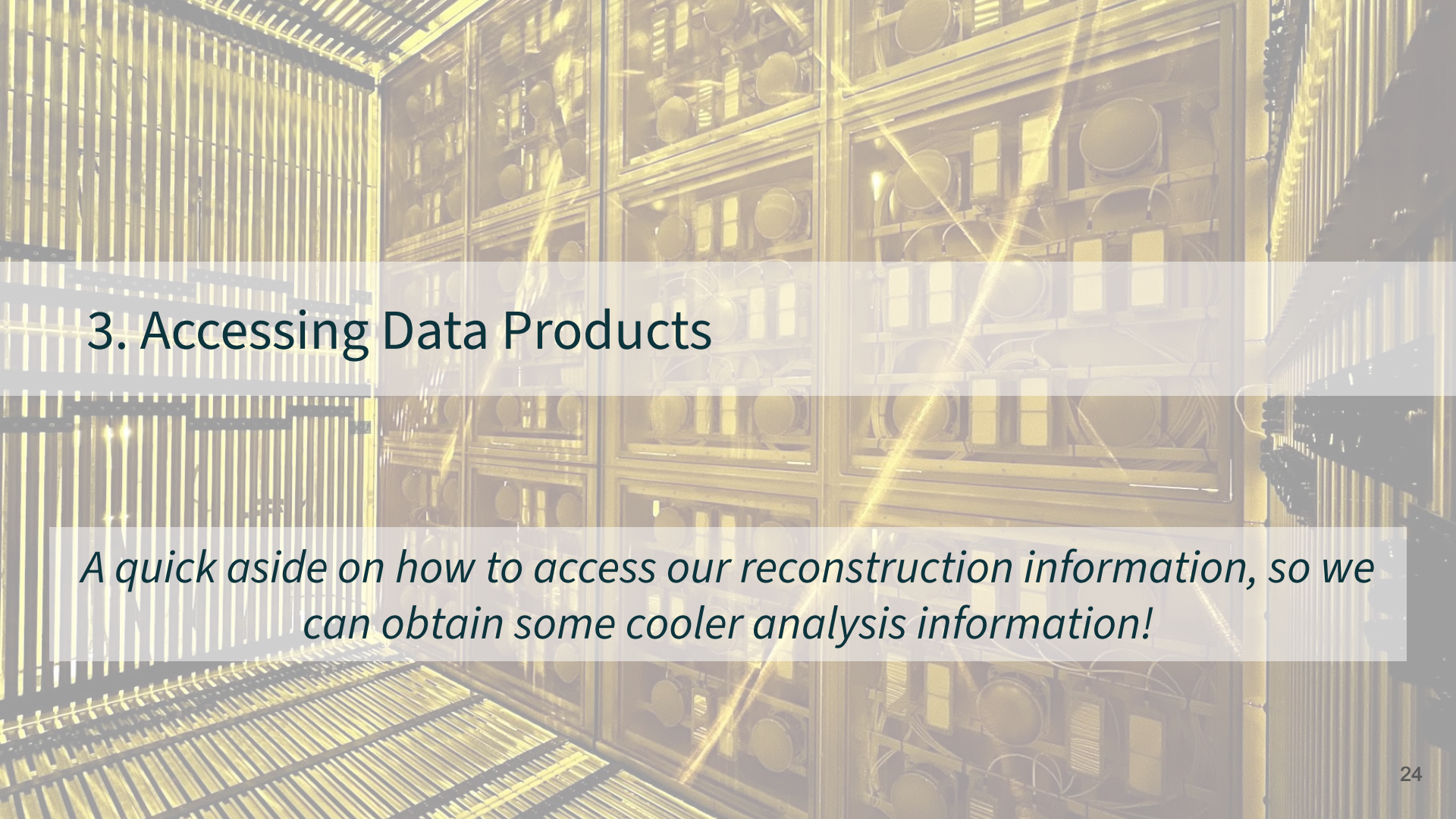
Looking at the output in ROOT

Here you can see that the name you gave to the analyzer in the fhicl run script is the name of your directory (**ana**): Open it with **->cd()**

You can see the output (T)Tree that we created, use **Scan()** to view its contents (can also use **Show(*entryNumber*)**, a **TBrowser** etc…)

Your tree exists and contains the event IDs! Success! (hopefully)





3. Accessing Data Products

A quick aside on how to access our reconstruction information, so we can obtain some cooler analysis information!

Accessing products from our files (1)

- Currently, just focused on EventID, but how do we access the information that we've added to the 'simulation/data' files e.g. in the Pandora stage?
- There are two ways the information is stored in these files:

```
std::vector<art::Ptr<recob::PFParticle>>

        {PFP_A,    PFP_B,    PFP_C}
```

1) As a vector of objects:

e.g. a vector of all `PFParticles` created by Pandora

```
{PFP_A → Vtx_B,
 PFP_B → Vtx_A,
 PFP_C → Vtx_C}
```

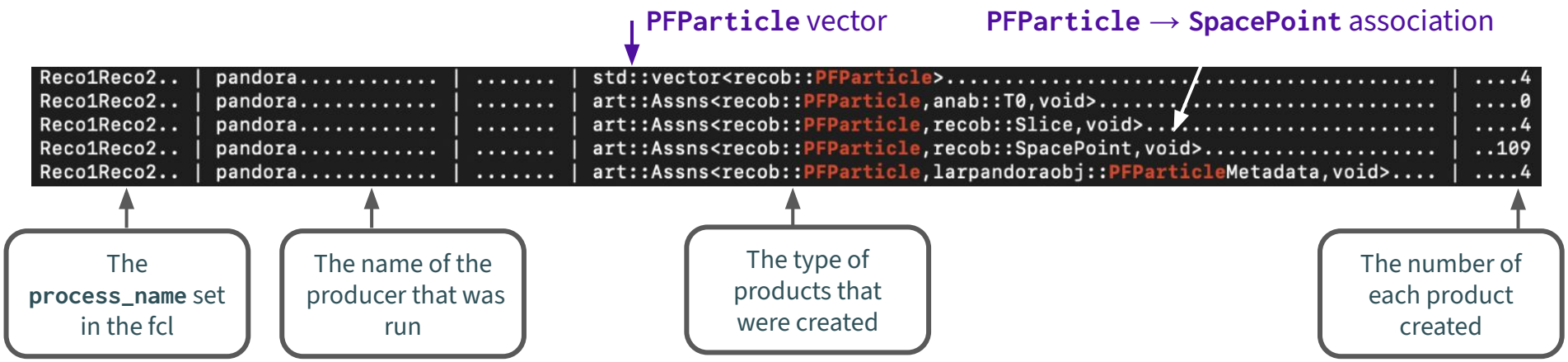
2) As associations:

e.g. links between `PFParticles` and their associated reconstructed vertex

Accessing products from our files (2)

- We can use eventdump.fcl to see what data products are saved in our ‘simulation/data’ files

```
lar -c eventdump.fcl whateverYourSimulationOrDataFileIsCalled.root -n 1
```



Accessing Vectors (the technical details)

- In our analyzer, let's say that we want to obtain the vector of slices
- We first need to set up the **data object handle**, consider this to be the link between your code and the object vector in the simulation/data files

```
art::ValidHandle<std::vector<recob::Slice>> sliceHandle = e.getValidHandle<std::vector<recob::Slice>>("pandora");
```

the type of object we're after

e is the current art::Event

the name of the producer that created it (see previous slide)

- After we check that our handle is valid, we can now retrieve the vector in our code

```
std::vector<art::Ptr<recob::Slice>> sliceVector;

if (sliceHandle.isValid())
    art::fill_ptr_vector(sliceVector, sliceHandle);
```

Accessing Associations (Technical Details)

- Say that, in our analyser, we want to obtain the vector of `PFParticles` connected to a given slice



- We first initialise a `FindManyP` object, consider this to be a link between your code and the associations of a given object vector (in this case, the vector in which our considered slice lives)

```
// Get associations between slices and pfparticles
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, "pandora");
```

our handle to the object vector

the name of the producer that created the association

Accessing Associations (Technical Details)

- To get the `PFParticles` associated to a particular slice, in this case the first slice in `sliceVector`
- We then do:

```
art::Ptr<recob::Slice> slice(sliceVector.at(0));

std::vector<art::Ptr<recob::PFParticle>> slicePFPs(slicePFPAssoc.at(slice.key()));
```

What's that **key** function about?

What's the key function about?

- Every `art::Ptr<...>` has a key function
- It returns the index of the 'pointed to' object in the vector in which it lives, and is used to identify the connected associations

Consider:

```
std::vector<art::Ptr<recob::Slice>> isobelsAwesomeSliceVector = {sliceA, sliceB, sliceC};
```

Then:

```
sliceA->key() == 0    sliceB->key() == 1    sliceC->key() == 2
```

So, to get the `PFParticle` vector associated with `sliceC`, we'd do:

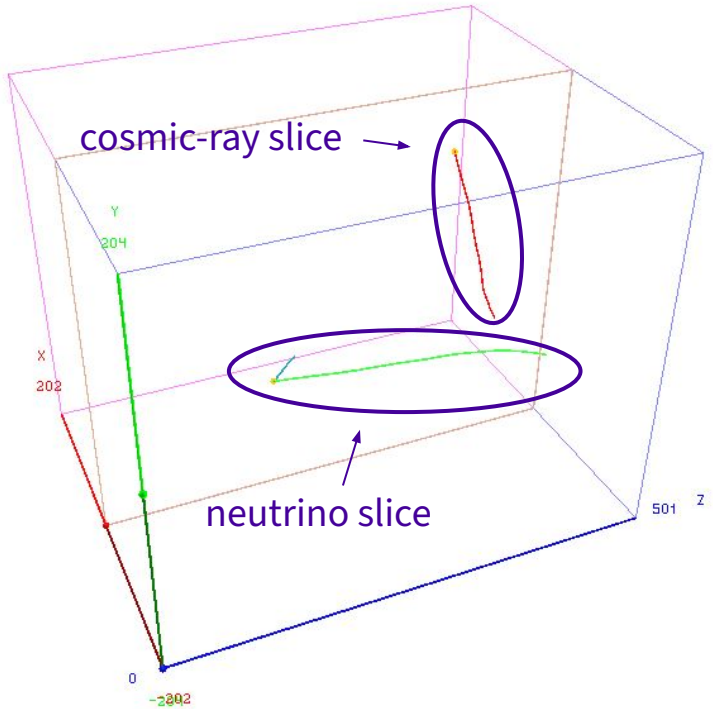
```
std:::vector<art::Ptr<recob::PFParticle>> slicePFPs = slicePFPAssoc.at(sliceC.key());
```



4. Investigating our Neutrino Hierarchy

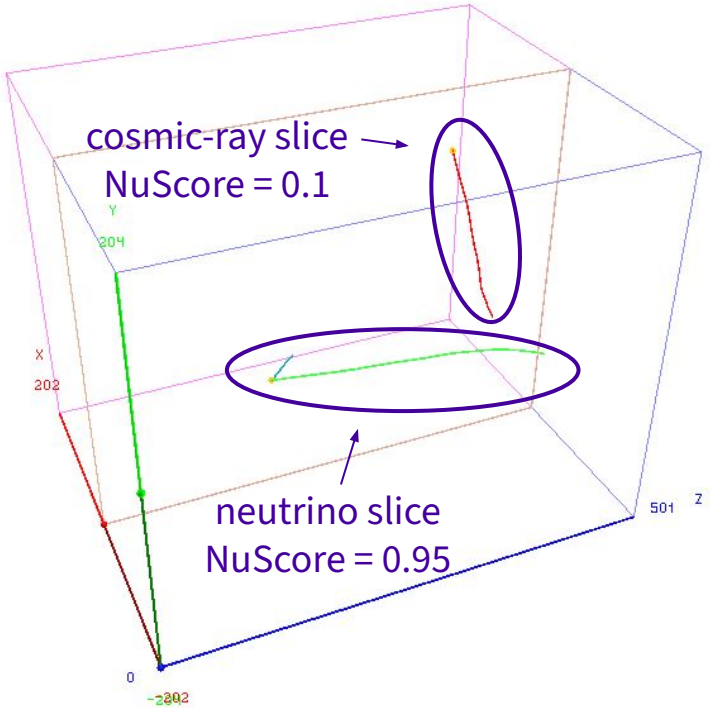
Obtaining the Neutrino Hierarchy (1)

- We've simulated a neutrino-interaction (1mu+1p) alongside a cosmic-ray muon
- Pandora will 'slice' up the input hits into groups containing a single cosmic-ray or neutrino interaction
- It will reconstruct each slice under a neutrino and cosmic-ray hypothesis
- In our SBND files, the neutrino reconstruction for each slice is saved



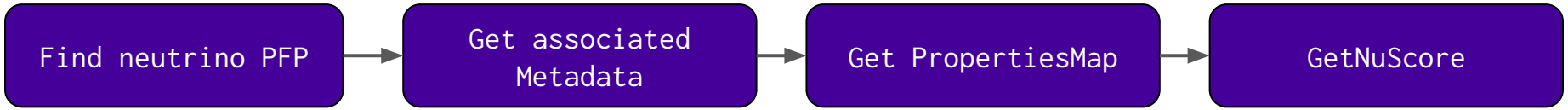
Obtaining the Neutrino Hierarchy (2)

- How do we identify our neutrino interactions?
- Pandora uses a BDT to assign each slice a score from 0 to 1, indicating how neutrino-like it is, with 1 being very neutrino-like
- This BDT uses topological information alone, experiments also incorporate light information e.g. CRUMBS on SBND



Neutrino Score

- So how do we access this neutrino score?
- For each PFParticle, Pandora output an **associated** larpandoraobj::PFParticleMetadata object
- This object contains a ‘PropertiesMap’ which stores some extra information about the PFP e.g. a score denoting how track-like the PFP is
- The neutrino PFP will hold the NuScore in this map



- In our example should have two slices: one containing the neutrino interaction and the other a cosmic-ray interaction
- We’ll identify the neutrino slice as that with the highest NuScore

Obtaining the Neutrino Hierarchy (3)

```
float highestNuScore = std::numeric_limits<float>::lowest();
for (const art::Ptr<recob::Slice> &slice : sliceVector)
{
  std::vector<art::Ptr<recob::PFParticle>> slicePFPs(slicePFPAssoc.at(slice.key()));
  for (const art::Ptr<recob::PFParticle> &slicePFP : slicePFPs)
  {
    const bool isPrimary(slicePFP->IsPrimary());
    const bool isNeutrino((std::abs(slicePFP->PdgCode()) == 12) || (std::abs(slicePFP->PdgCode()) == 14));
    if (!isPrimary && !isNeutrino) { continue; }

    // Get NuScore
    std::vector<art::Ptr<larpandoraobj::PFParticleMetadata>> pfpMetaVec = pfpMetaDataAssoc.at(slicePFP->Self());
    art::Ptr<larpandoraobj::PFParticleMetadata> pfpMeta = pfpMetaVec.front();
    larpandoraobj::PFParticleMetadata::PropertiesMap propertiesMap = pfpMeta->GetPropertiesMap();
    auto propertiesMapIter = propertiesMap.find("NuScore");

    if (propertiesMapIter != propertiesMap.end())
    {
      const float thisNuScore(propertiesMapIter->second);
      if (thisNuScore > highestNuScore)
      {
        highestNuScore = thisNuScore;
        // We have found our neutrino!
        nuSliceKey = slice.key();
        nuID = slicePFP->Self();
      }
    }
  }
}
```

The neutrino PFP stores the NuScore, so first find the 'primary' PFP in the slice

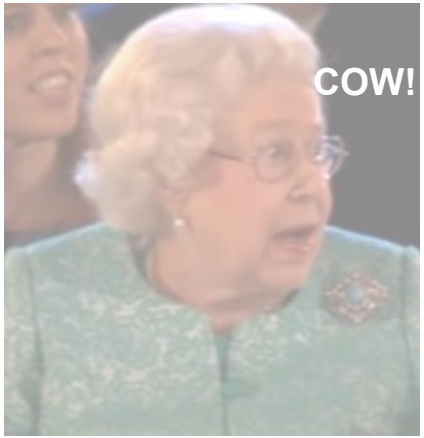
The NuScore is stored in the property map of the PFP's 'Metadata'

Save the 'key' and ID of the highest NuScore neutrino so that we can access it later!

The Neutrino Hierarchy in LArSoft

```
didILeaveTheOvenOnPFP->Self() == 5
didILeaveTheOvenOnPFP->Parent() == 11
didILeaveTheOvenOnPFP->Daughter() == {4, 13}
```

```
curiousGeorgePFP->Self() == 3
curiousGeorgePFP->Parent() == 13
curiousGeorgePFP->Daughters() == {}
```









```
queeniePFP->Self() == 11
queeniePFP->Parent() == kPFParticlePrimary
queeniePFP->Daughters() == {1, 5, 7, 9}
```

```
willIAmPFP->Self() == 13
willIAmPFP->Parent() == 5
willIAmPFP->Daughters() == {3, 10, 6}
```

Implementing Neutrino Hierarchy Variables (1)

- Let's 'calculate' some neutrino hierarchy variables, and add them to our tree!

1) First, we'll need some new includes:

```
#include "lardataobj/RecoBase/Slice.h"
#include "lardataobj/RecoBase/PFParticle.h"
#include "lardataobj/RecoBase/PFParticleMetadata.h"
#include "canvas/Persistency/Common/FindManyP.h"
```

2) Create new member variables, and connect them to our (T)Tree

In the private section of our class definition

```
// Tree variables
unsigned int fEventID;
unsigned int fNPFParticles;
unsigned int fNPrimaryChildren;
```

In beginJob

```
// Add branches to TTree
fTree->Branch("eventID", &fEventID);
fTree->Branch("nPFParticles", &fNPFParticles);
fTree->Branch("nPrimaryChildren", &fNPrimaryChildren);
```

3) Calculate the neutrino hierarchy variables (1)

In analyze

```
// Set the event ID
fEventID = e.id().event();

// Prepare variables for new event
fNPFParticles = 0;
fNPrimaryChildren = 0;

// Get event slices
art::ValidHandle<std::vector<recob::Slice>> sliceHandle = e.getValidHandle<std::vector<recob::Slice>>("pandora");
std::vector<art::Ptr<recob::Slice>> sliceVector;
if (sliceHandle.isValid())
  art::fill_ptr_vector(sliceVector, sliceHandle);


// Get associations (Slice->PFP) & (PFP->Metadata)
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, "pandora");
art::ValidHandle<std::vector<recob::PFParticle>> pfpHandle = e.getValidHandle<std::vector<recob::PFParticle>>("pandora");
art::FindManyP<larpandoraobj::PFParticleMetadata> pfpMetaDataAssoc(pfpHandle, e, "pandora");
```

Initialise variables to zero at the start of every event, as they will still hold the value from the previous event!

Collect the slices

Setup the slice→PFP and PFP→Metadata associations

We'll need to get the PFParticle handle so that we can initialise the FindManyP object

3) Calculate the neutrino hierarchy variables (2)

After associations

```
[[maybe_unused]] int nuID = -1, nuSliceKey = -1;
float highestNuScore = std::numeric_limits<float>::lowest();
for (const art::Ptr<recob::Slice> &slice : sliceVector)
{
  std::vector<art::Ptr<recob::PFParticle>> slicePFPs(slicePFPAssoc.at(slice.key()));
  for (const art::Ptr<recob::PFParticle> &slicePFP : slicePFPs)
  {
    const bool isPrimary(slicePFP->IsPrimary());
    const bool isNeutrino((std::abs(slicePFP->PdgCode()) == 12) || (std::abs(slicePFP->PdgCode()) == 14));
    if (!isPrimary || !isNeutrino) { continue; }

    // Get NuScore
    std::vector<art::Ptr<larpandoraobj::PFParticleMetadata>> pfpMetaVec = pfpMetaDataAssoc.at(slicePFP->Self());
    art::Ptr<larpandoraobj::PFParticleMetadata> pfpMeta = pfpMetaVec.front();
    larpandoraobj::PFParticleMetadata::PropertiesMap propertiesMap = pfpMeta->GetPropertiesMap();
    auto propertiesMapIter = propertiesMap.find("NuScore");

    if (propertiesMapIter != propertiesMap.end())
    {
      float thisNuScore(propertiesMapIter->second);
      if (thisNuScore > highestNuScore)
      {
        highestNuScore = thisNuScore;
        nuSliceKey = slice.key();
        nuID = slicePFP->Self();
        fNPFParticles = slicePFPs.size();
        fNPrimaryChildren = slicePFP->NumDaughters();
      }
    }
  }
}
```

Initialise our neutrino identifier variables to something nonsensical

Our loop from earlier!

Set the neutrino hierarchy variables, these will be overwritten such that in the end they correspond to the slice with the highest NuScore

3) Calculate the neutrino hierarchy variables (3)

After neutrino slice found

```
// If we haven't found a neutrino, move on
if (nuID < 0)
  return;

// Fill tree
fTree->Fill();

} // analyze()
```

If neutrino identifier variables still equal a nonsensical value, we didn't find a neutrino slice, so skip event!

Fill the (T)Tree now that all our variables have been set

HARD CODING MODULE NAMES IS A VERY VERY VERY BAD IDEA!



We’ve been lazy and done this:

```
// Get associations (Slice->PFP) & (PFP->Metadata)
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, 'pandora');
art::FindManyP<larpandoraobj::PFParticleMetadata> pfpMetaDataAssoc(pfpHandle, e, 'pandora');
```



This is a really dumb idea, and we should instead do this:

```
// Get associations (Slice->PFP) & (PFP->Metadata)
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);
art::FindManyP<larpandoraobj::PFParticleMetadata> pfpMetaDataAssoc(pfpHandle, e, fMetaDataLabel);
```

Implementing Neutrino Hierarchy Variables (4)

- But how do we set fSliceLabel and fMetaDataLabel?
- We pass module names into our analyzer through the `analysisConfig.fcl` file:
- This way they're configurable, so if we (or a user of your module) needs to change them we/they can!

In your analyzer:

```
In the private section of our class definition

// Define input labels
std::string fSliceLabel;
std::string fPFParticleLabel;
std::string fMetaDataLabel;
```

```
test::AnalyseEvents::AnalyseEvents(fhicl::ParameterSet const& p)
 : EDAnalyzer{p},
 fSliceLabel(p.get<std::string>("SliceLabel")),
 fPFParticleLabel(p.get<std::string>("PFParticleLabel")),
 fMetaDataLabel(p.get<std::string>("MetaDataLabel"))
 {
   // Call appropriate consumes<>() …
 }
```

In `analysisConfig.fcl`:

```
BEGIN_PROLOG

analyseEvents:
{
  module_type:  "AnalyseEvents"

  SliceLabel:        "pandora"
  PFParticleLabel:   "pandora"
  MetaDataLabel:     "pandora"
}

END_PROLOG
```

Implementing Neutrino Hierarchy Variables (5)

- And let's remove our hardcoded values

In analyze

```
// Get event slices
art::ValidHandle<std::vector<recob::Slice>> sliceHandle = e.getValidHandle<std::vector<recob::Slice>>("pandora");

...

// Get associations (Slice->PFP) & (PFP->Metadata)
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, "pandora");
art::ValidHandle<std::vector<recob::PFParticle>> pfpHandle = e.getValidHandle<std::vector<recob::PFParticle>>("pandora");
art::FindManyP<larpandoraobj::PFParticleMetadata> pfpMetaDataAssoc(pfpHandle, e, "pandora");
```



```
// Get event slices
art::ValidHandle<std::vector<recob::Slice>> sliceHandle = e.getValidHandle<std::vector<recob::Slice>>(fSliceLabel);

...

// Get associations (Slice->PFP) & (PFP->Metadata)
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);
art::ValidHandle<std::vector<recob::PFParticle>> pfpHandle = e.getValidHandle<std::vector<recob::PFParticle>>(fPFParticleLabel);
art::FindManyP<larpandoraobj::PFParticleMetadata> pfpMetaDataAssoc(pfpHandle, e, fMetaDataLabel);
```

Fhicl configuration file linking & running

```
source build.sh
```

Compile changes

```
lar -c run_analyseEvents.fcl -s /path/to/input/file.root -n 10
```

Run analyzer

```
root -l analysisOutput.root
```

Check output

44

What our output looks like now

- Our (T)Tree should now have 2 new branches

`nPFParticles` tells us how many particle we have reconstructed

`nPrimaryChildren` is the number of primary particles (children of the neutrino) we have reconstructed

- By viewing the tree, we can check that everything looks sensible…

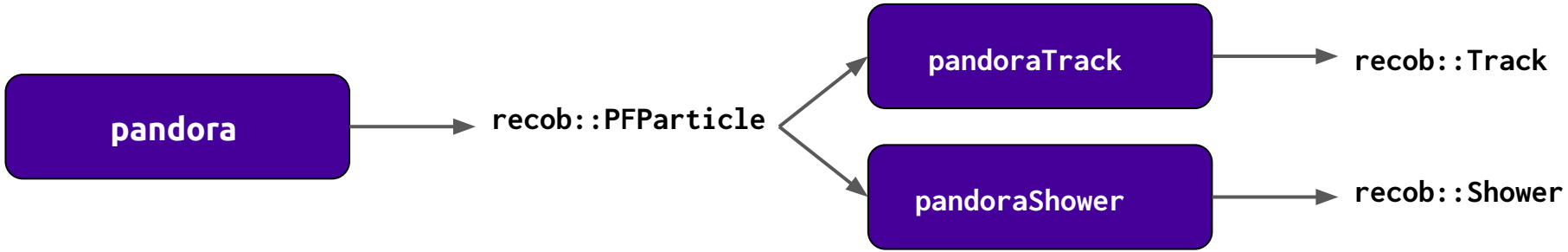
```
[Analysis > root -l analysisOutput.root
 root [0]
 Attaching file analysisOutput.root as _file0...
 (TFile *) 0x30c2b70
[root [1] .ls
 TFile**         analysisOutput.root
  TFile*         analysisOutput.root
   KEY: TDirectoryFile   ana;1   ana (AnalyzeEvents) folder
[root [2] ana->cd()
 (bool) true
[root [3] .ls
 TDirectoryFile*         ana     ana (AnalyzeEvents) folder
  KEY: TTree     tree;1  Output TTree
[root [4] tree->Scan()
**********************************************************************
*    Row   * eventID.e * nPFPartic * nPrimaryC *
**********************************************************************
*        0 *         1 *         3 *         2 *
*        1 *         2 *         6 *         5 *
*        2 *         3 *         4 *         3 *
*        3 *         4 *         3 *         2 *
*        4 *         5 *         4 *         3 *
*        5 *         6 *         3 *         2 *
*        6 *         7 *         5 *         3 *
*        7 *         8 *         4 *         2 *
*        8 *         9 *         4 *         2 *
*        9 *        10 *         4 *         2 *
**********************************************************************
```



5. Adding Track Information

Let's have a look at the length of our muon/proton tracks

In the SBND workflow, all PFParticles are fitted as both tracks and showers



- The association we are after is:

`recob::PFParticle → recob::Track`

- Let's grab these tracks and have a look at some track related variables!

Registering our Track Producer Module (1)

```
#include "lardataobj/RecoBase/Track.h"
```

Add relevant header

```
std::string fTrackLabel;
```

Add new member variable to store the TrackLabel set in fcl

```
test::AnalyseEvents::AnalyseEvents(fhicl::ParameterSet const& p)
 : EDAnalyzer{p} ,
 fSliceLabel(p.get<std::string>("SliceLabel")),
 fPFParticleLabel(p.get<std::string>("PFParticleLabel")),
 fMetaDataLabel(p.get<std::string>("MetaDataLabel")) ,
 fTrackLabel(p.get<std::string>("TrackLabel"))
 {
   // Call appropriate consumes<>() ...
 }
```

Initialise TrackLabel

Look back on the previous exercise if you're not sure where each bit goes

Registering our Track Producer Module (2)

- In the configuration file (analysisConfig.fcl) we need to set the track producer label

```
BEGIN_PROLOG

analyseEvents:
{
  module_type:  "AnalyseEvents"

  SliceLabel:       "pandora"
  PFParticleLabel:  "pandora"
  TrackLabel:       "pandoraTrack"
  MetaDataLabel:    "pandora"
}

END_PROLOG
```

Creating the Track Related Variable

```
std::vector<float> fChildTrackLengths;
```

Add a new output to store the lengths of reconstructed tracks!

```
fChildTrackLengths.clear();
```

Reset the vector for each analysed event

```
fTree->Branch("childTrackLengths", &fChildTrackLengths);
```

Add a new branch to our (T)Tree

Look back on the previous exercise if you're not sure where each bit goes

Recording Track Length

After nuID < 0 check

```
// Store neutrino's PFPs
 std::vector<art::Ptr<recob::PFParticle>> nuSlicePFPs(slicePFPAssoc.at(nuSliceKey));

// Now let's look at our tracks
art::FindManyP<recob::Track> pfpTrackAssoc(pfpHandle, e, fTrackLabel);
for (const art::Ptr<recob::PFParticle> &nuSlicePFP : nuSlicePFPs)
{
   // Only interested in neutrino children particles
   if (nuSlicePFP->Parent() != static_cast<long unsigned int>(nuID))
     continue;

   // Get tracks associated with this PFParticle
   std::vector<art::Ptr<recob::Track>> tracks = pfpTrackAssoc.at(nuSlicePFP.key());

   // There should only be 0 or 1 tracks associated with a PFP
   if (tracks.size() != 1)
     continue;

   // Get the track
   art::Ptr<recob::Track> track = tracks.at(0);

   // Add parameters from the track to the branch vector
   fChildTrackLengths.push_back(track->Length());
}
```



Another way to view our analysis results…

Let's look at the track lengths

You can also use `-n -1`

Firstly, run over all your events by removing `-n 10` from the command like this:

```
lar -c run_analyseEvents.fcl -s /path/to/input/file.root
```

Open the output file and draw the track lengths! (using `treeName->Draw("branch name")`)

```
root -l analysisOutput.root
```

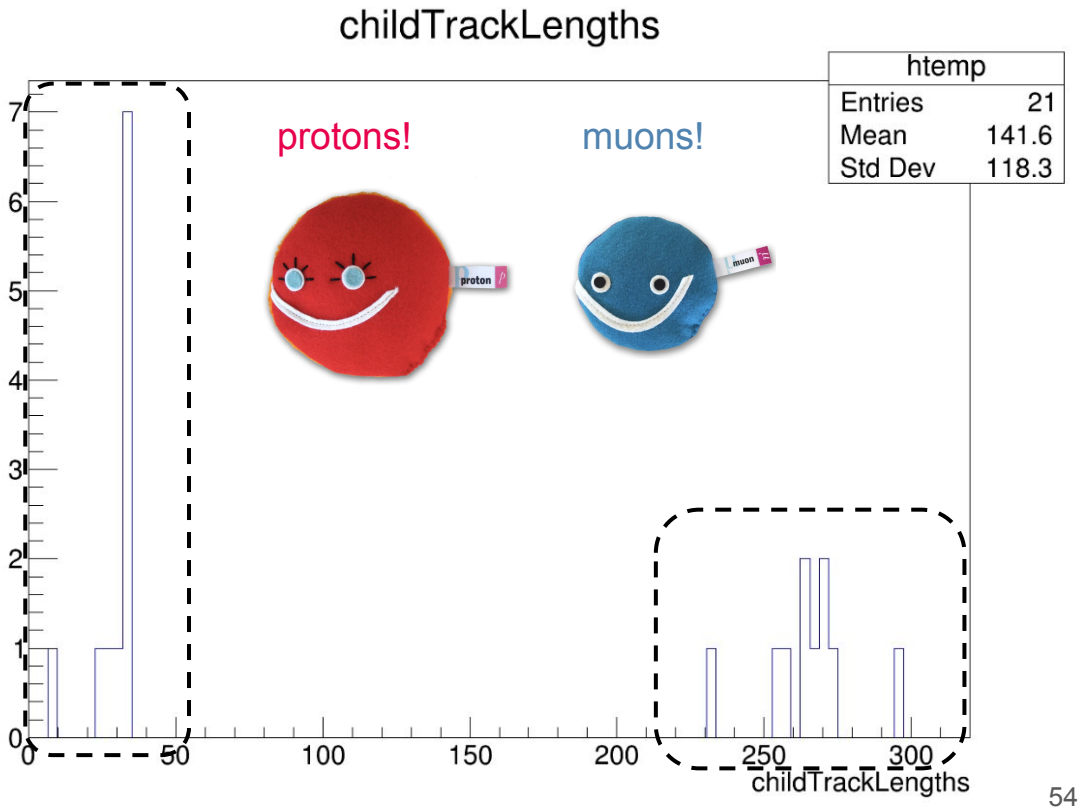
On the terminal

```
root[0] ana->cd()
root[1] tree->Draw("childTrackLengths")
```

In the root terminal

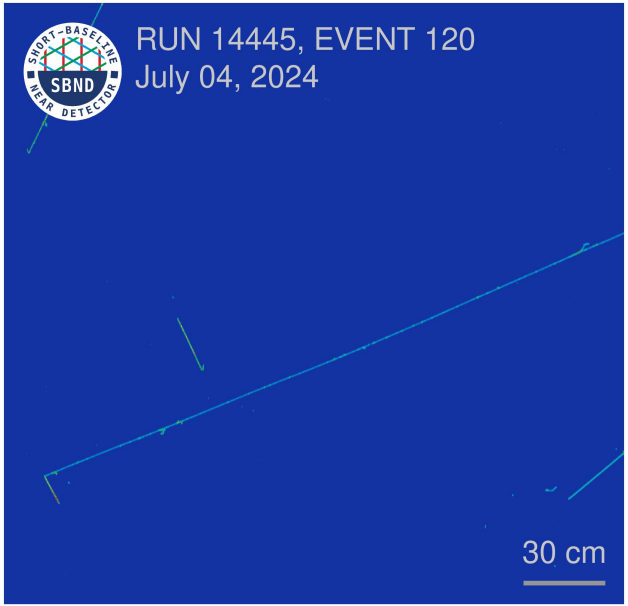
What do you see?

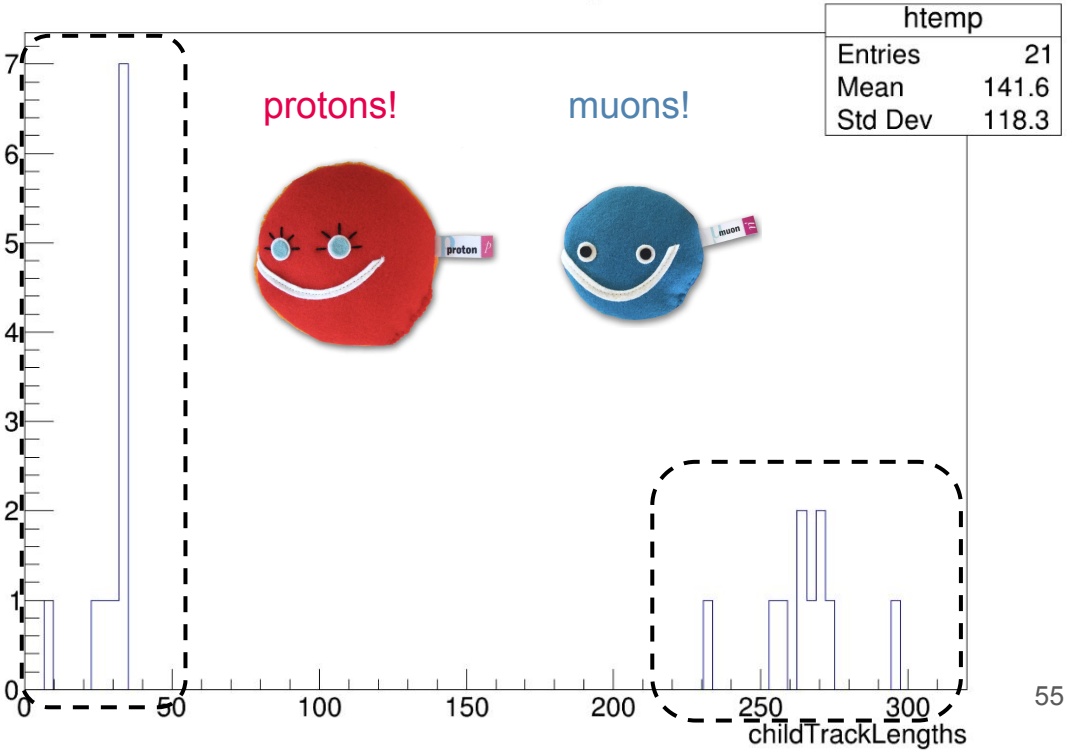
- We can clearly make out two areas corresponding to our two particles
- **But where do the protons live and where do the muons live?**



What do you see?

- And which is which in the event display?





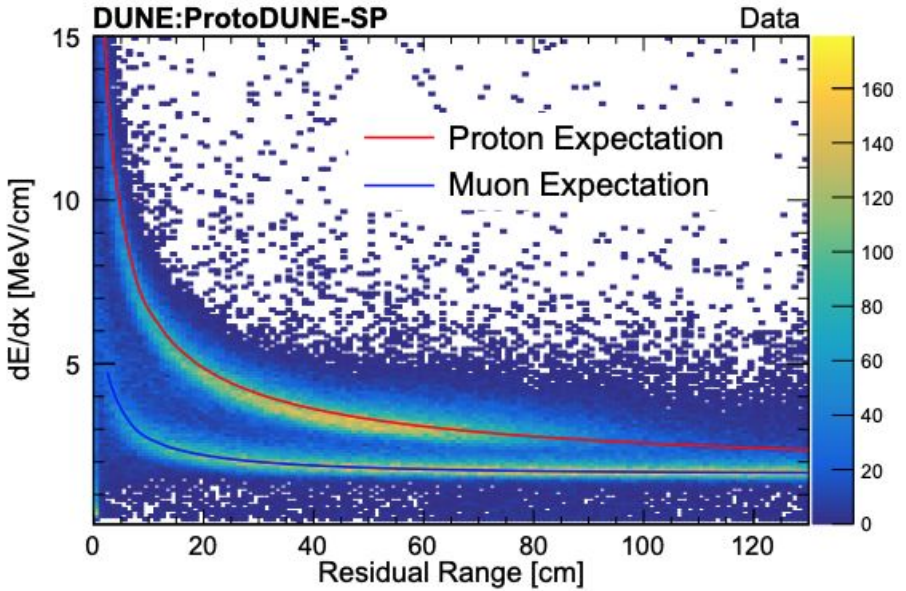


7. Associations: Going a little deeper

Particle Ionisation

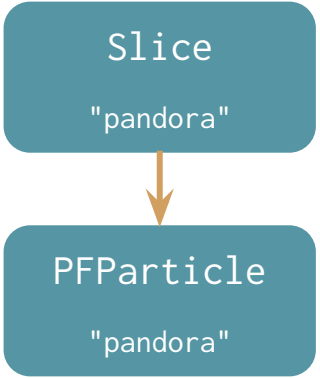
- Take a look at this plot from ProtoDUNE-SP showing the dE/dx vs. residual range distributions for muons and protons produced in a test beam at CERN
- The theoretical distributions are overlaid
- We see that protons deposit more energy as a function of distance due to their larger mass
- Can we recreate this plot?

[2007.06722] First results on ProtoDUNE-SP...



More associations!

Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s

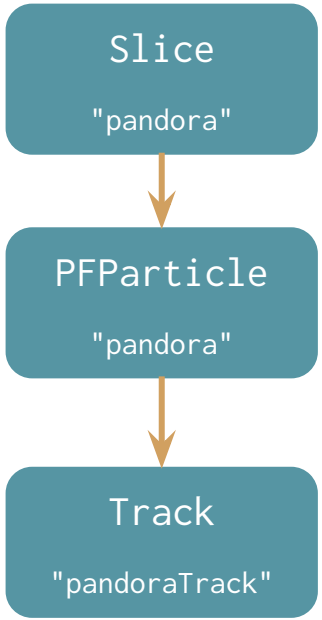


More details can be found in the doxygen entry.

More associations!

Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s

…and then between `recob::PFParticle`s and `recob::Track`s.



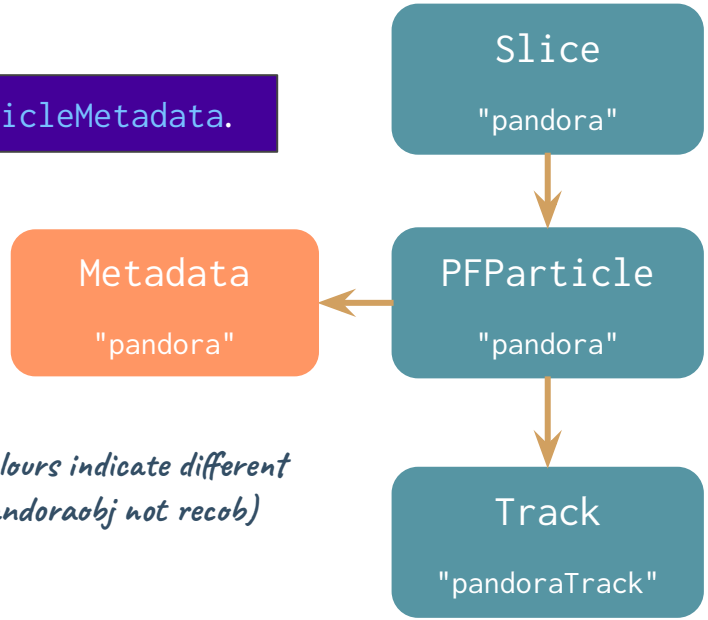
More details can be found in the doxygen entry.

More associations!

Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s

…and then between `recob::PFParticle`s and `recob::Track`s.

…and then between `recob::PFParticle`s and `larpandoraobj::PFParticleMetadata`.



Notice different colours indicate different namespaces (larpandoraobj not recob)

More details can be found in the doxygen entry.

More associations!

Notice different colours indicate different namespaces (recob, larpandoraobj & anab)

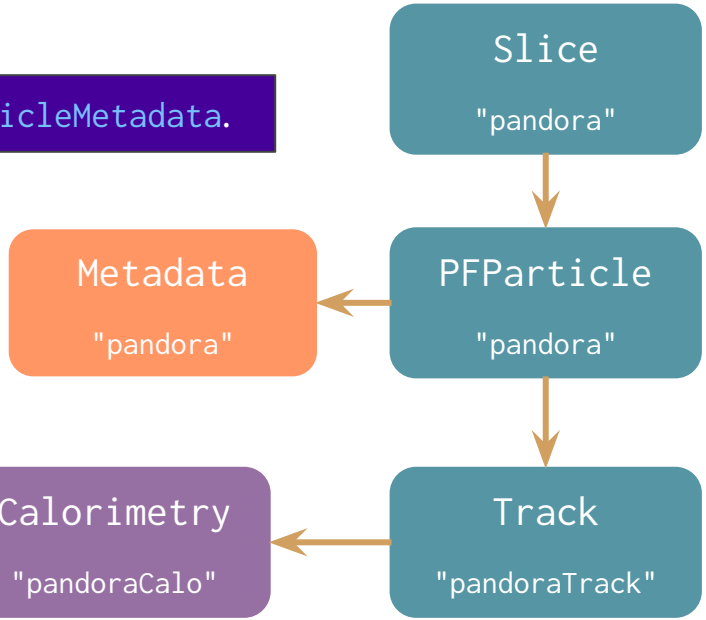
Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s

…and then between `recob::PFParticle`s and `recob::Track`s.

…and then between `recob::PFParticle`s and `larpandoraobj::PFParticleMetadata`.

…we can now make use of another association to get hold of the energy deposition information we need to to recreate that ProtoDUNE plot.

This time we need the `anab::Calorimetry` object…



More details can be found in the doxygen entry.

More associations!

Notice different colours indicate different namespaces (recob, larpandoraobj & anab)

Earlier we looked at the association between `recob::Slices` and `recob::PFParticles`

…and then between `recob::PFParticles` and `recob::Tracks`.

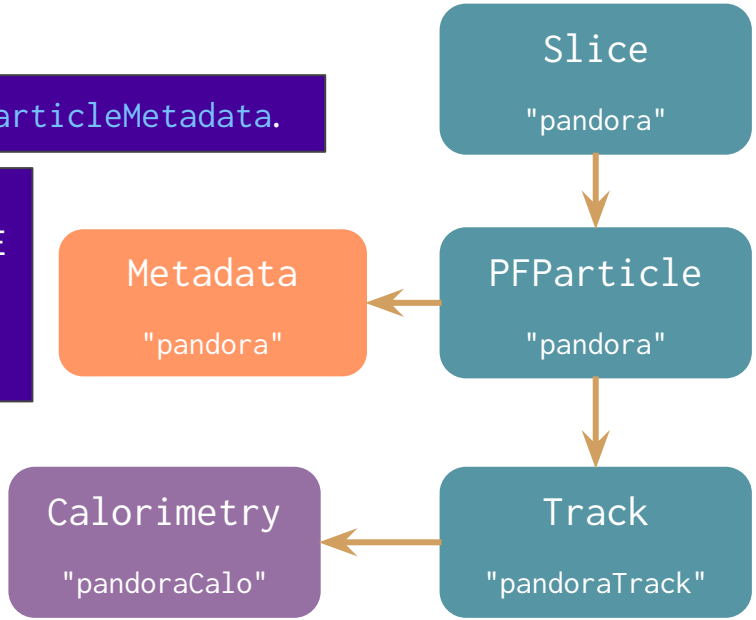
…and then between `recob::PFParticles` and `larpandoraobj::PFParticleMetadata`.

…we can now make use of another association to get hold of the energy deposition information we need to to recreate that ProtoDUNE plot.

This time we need the `anab::Calorimetry` object…

We have at least one separate calorimetry object for each of the three planes

The object contains vectors of dQ/dx, dE/dx, Residual Range etc values. Each entry corresponds to a trajectory point.



More details can be found in the doxygen entry.

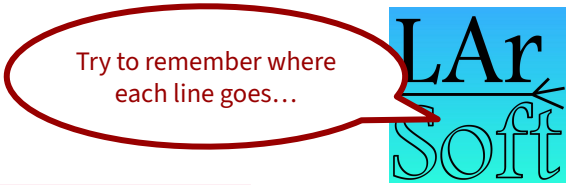
Accessing Calorimetry

These steps should feel familiar:

1. Add the relevant header for the `anab::Calorimetry` object
2. Add the module label to your configuration file and access it in the constructor
3. Add any declarations & branches for new variables you want to push to your tree
4. Access the list of `anab::Calorimetry` objects from a list of `recob::Track` objects using `art::FindManyP`
5. Fill your tree variables with information from your `anab::Calorimetry` object.

Try making a start on this and we'll go through it in more detail in a few minutes…

Accessing Calorimetry (1)



1. Add the relevant header for the `anab::Calorimetry` object

```
#include "lardataobj/AnalysisBase/Calorimetry.h"
```

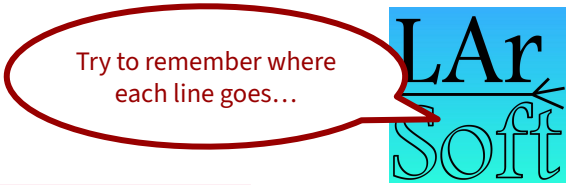
2. Add the module label to your configuration file and access it in the constructor

```
 std::string fCalorimetryLabel;
```

```
fCalorimetryLabel(p.get<std::string>("CalorimetryLabel"))
```

```
  CalorimetryLabel: "pandoraCalo"
```

Accessing Calorimetry (1)



3. Add any declarations & branches for new variables you want to push to your tree

```
std::vector<std::vector<float>> fChildTrackdEdx;
std::vector<std::vector<float>> fChildTrackResRange;
```

```
fTree->Branch("childTrackdEdx", &fChildTrackdEdx);
fTree->Branch("childTrackResRange", &fChildTrackResRange);
```

4. Access the list of `anab::Calorimetry` objects from a list of `recob::Track` objects using `art::FindManyP`

```
art::ValidHandle<std::vector<recob::Track>> trackHandle = e.getValidHandle<std::vector<recob::Track>>(fTrackLabel);
```

```
art::FindManyP<anab::Calorimetry> trackCaloAssoc(trackHandle, e, fCalorimetryLabel);
```

Accessing Calorimetry (2)

5. Fill your tree variables with information from your `anab::Calorimetry` object.

```
// Get the calorimetry object
std::vector<art::Ptr<anab::Calorimetry>> calos = trackCaloAssoc.at(track.key());
for(auto const& calo : calos)
{
  const int plane = calo->PlaneID().Plane;

  // Only interested in the collection plane (2)
  if(plane != 2)
    continue;

  fChildTrackdEdx.push_back(calo->dEdx());
  fChildTrackResRange.push_back(calo->ResidualRange());
}
```

Remember, there's (hopefully) a Calorimetry object for each readout plane

Histogram time!

You should be pretty familiar with rebuilding & running your analyzer now…

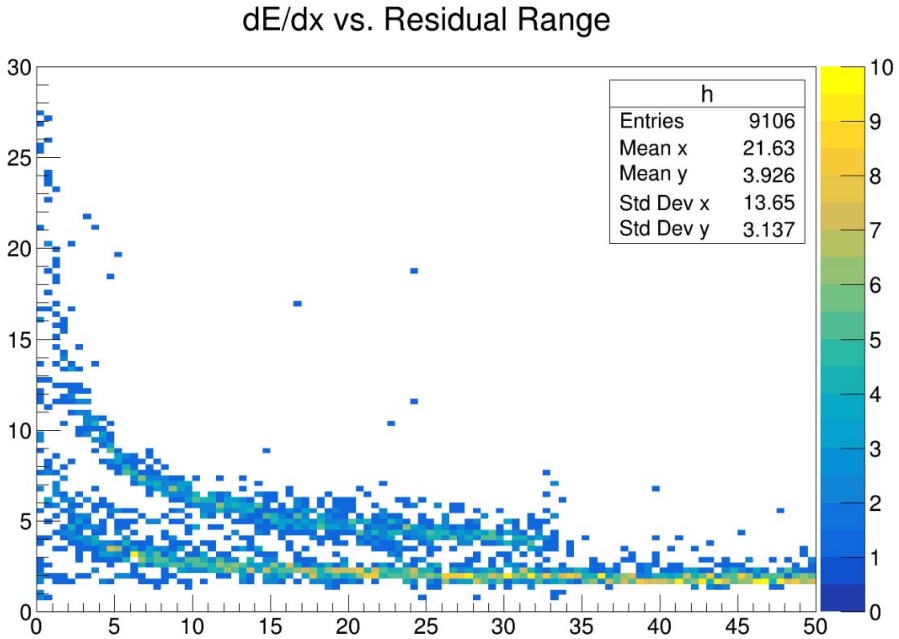
You can now use your calorimetry branches to make a 2D histogram in ROOT.

```
root[0] ana->cd()
```

```
root[1] TH2D *h = new TH2D("h","dE/dx vs. Residual Range", 200, 0, 50, 200, 0, 30)
```

```
root[2] tree->Draw("childTrackdEdx:childTrackResRange>>h", "", "colz")
```

You should see something like this!

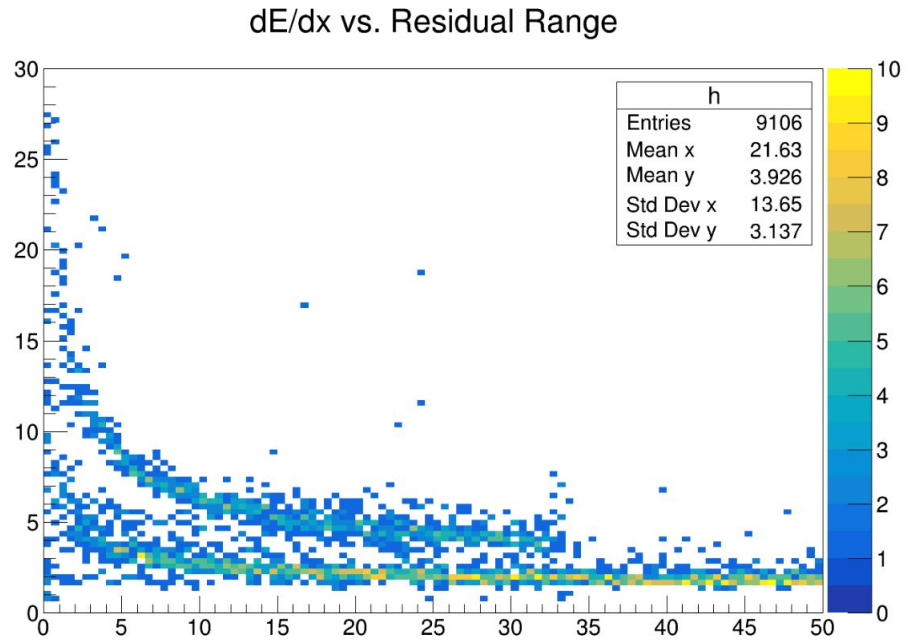


Try playing around with the axis labels/style options using the GUI.

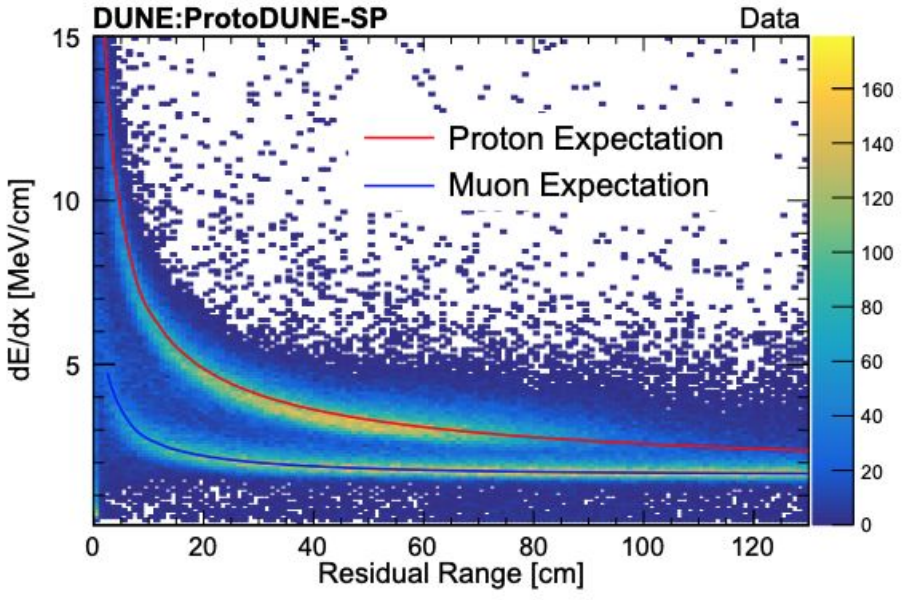
You can save the plot at the end too!

Muon/Proton dE/dx

- We've managed to recreate this plot!
- And we can clearly see two distinct contributions…



[2007.06722] First results on ProtoDUNE-SP...

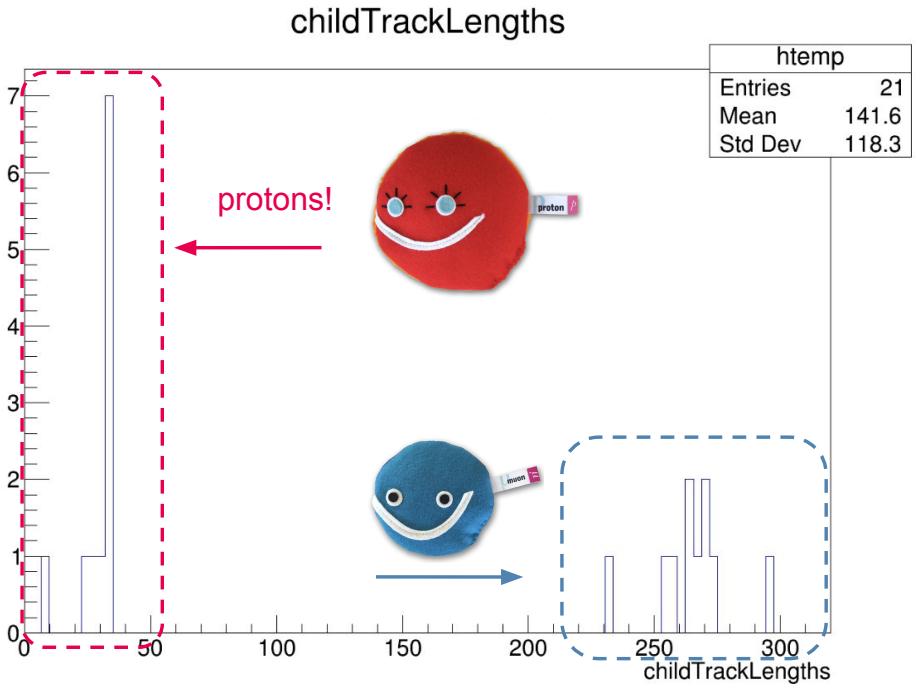




8. A **very** simple PID

A Simple PID

- Protons are highly ionising particles (HIPs), while muons are minimally ionising particles (MIPs)
- At the energies at which we've simulated our particles, our muon tracks should therefore be much longer than our proton tracks (we saw this earlier)
- We're going to build a very simple particle identification (PID) where we identify the muon as the longest track in our interaction
- To do this, we need to add to our code a loop over our neutrino children in which we identify the longest track



Finding the longest track (1)

```
// Let's find the longest track before we progress with filling the track variables
int longestID = std::numeric_limits<int>::lowest();
float longestLength = std::numeric_limits<float>::lowest();
for(const art::Ptr<recob::PFParticle> &nuSlicePFP : nuSlicePFPs)
{
  // We are only interested in neutrino children particles
  if (nuSlicePFP->Parent() != static_cast<long unsigned int>(nuID))
    continue;

  // Get tracks associated with this PFParticle
  std::vector<art::Ptr<recob::Track>> tracks = pfpTrackAssoc.at(nuSlicePFP.key());

  // There should only be 0 or 1 tracks associated with a PFP
  if (tracks.size() != 1) { continue; }

  // Get the track
  art::Ptr<recob::Track> track = tracks.at(0);

  // Check if this track is longer than the current longest
  if(track->Length() > longestLength)
  {
    longestID = track->ID();
    longestLength = track->Length();
  }
}
```

Keep track of the longest track and its length

Loop over the PFPs and access the associated tracks

Below should be our existing loop over the nuPFPs where we fill our tree variables

Check whether the current track replaces the current longest

Finding the longest track (2)

- Now, let's create and fill another tree variable to store whether each track is the longest in the interaction

1. We'll fill our tree variable like so:

```
fChildTrackIsLongest.push_back(track->ID() == longestID);
```

2. I'll leave you to initialise, reset, and link the above variable to our output tree

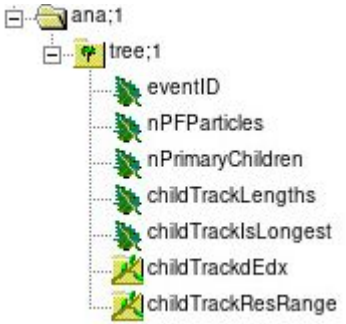
   - HINT: fChildTrackIsLongest should be a vector of booleans

Finding the longest track (3)

Once you think you have included all the necessary additions you will, as usual, need to recompile your analyzer and run it over your reconstruction file again…

Applying our PID (1)

- We can now make two plots, one corresponding to the dE/dx of the longest track and the other all other tracks (dominated by our proton)
- Let's open our file again, this time making two versions of our dE/dx vs. Residual Range histogram.



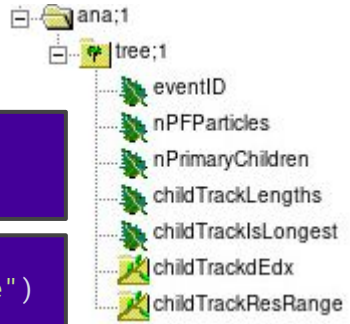

```
root[0] ana->cd()
```

```
root[1] TH2D *hLong = new TH2D("hLong","dE/dx vs. Residual Range", 200, 0, 50, 200, 0, 30)
```

```
root[2] TH2D *hShort = new TH2D("hShort","dE/dx vs. Residual Range", 200, 0, 50, 200, 0, 30)
```

Applying our PID (2)

This time we need to include our condition on the draw command…




```
root[3] tree->Draw("childTrackdEdx:childTrackResRange>>hLong", "childTrackIsLongest", "")
```

```
root[4] tree->Draw("childTrackdEdx:childTrackResRange>>hShort", "!childTrackIsLongest", "same")
```

We need to tell the two apart… Let's draw them in different colours!

```
root[5] hLong->SetMarkerColor(kMagenta+2)
```

Alternative colour options are here: https://root.cern.ch/doc/master/classTColor.html

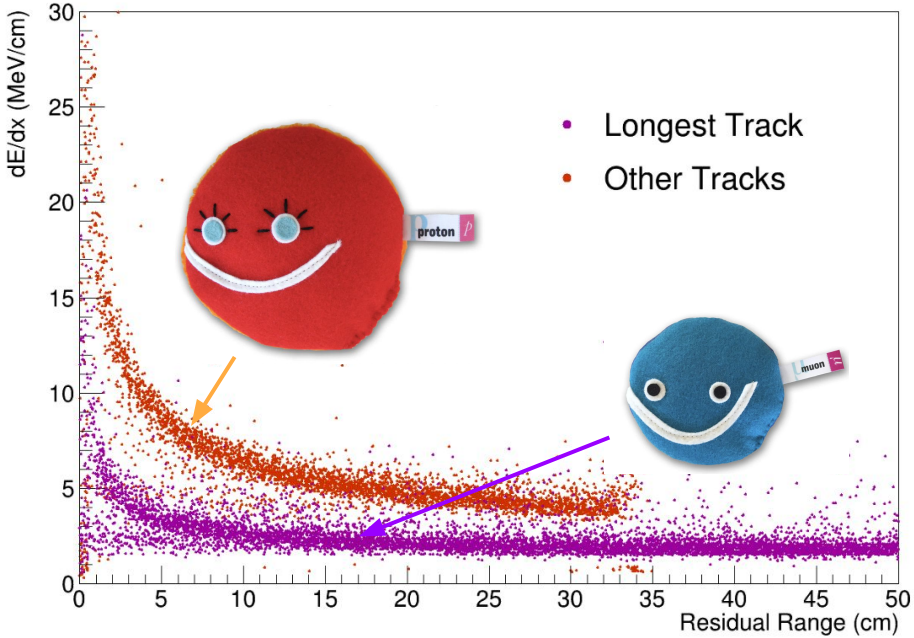
```
root[6] hShort->SetMarkerColor(kOrange+2)
```

```
root[6] c1->Modified()
```

Tell the canvas (default c1) to implement these changes and redraw the canvas

Energy deposition

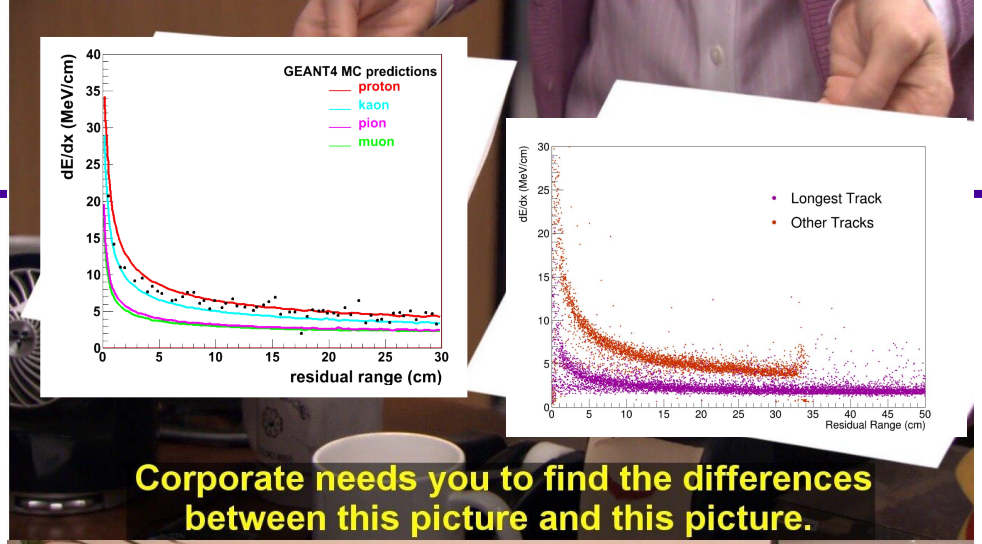
- We can see that our identified muons correspond to the MIP distribution, and our identifies protons correspond to the HIP distribution!
- It looks like our simple PID works quite well!
- Your plot will be a bit sparse, so run your analyser on this file to boost your stats:



`/Workshop/TPCSimulation/example_outputs/no_spread/reco2_100evts_nospread.root`

arXiv:1205.6747v2
[physics.ins-det] 5 Jun 2012

This ArgoNeuT plot shows the theoretical separating power of the average dE/dx vs. residual range distributions. The overlaid black data points show a single stopping track in the ArgoNeuT detector.

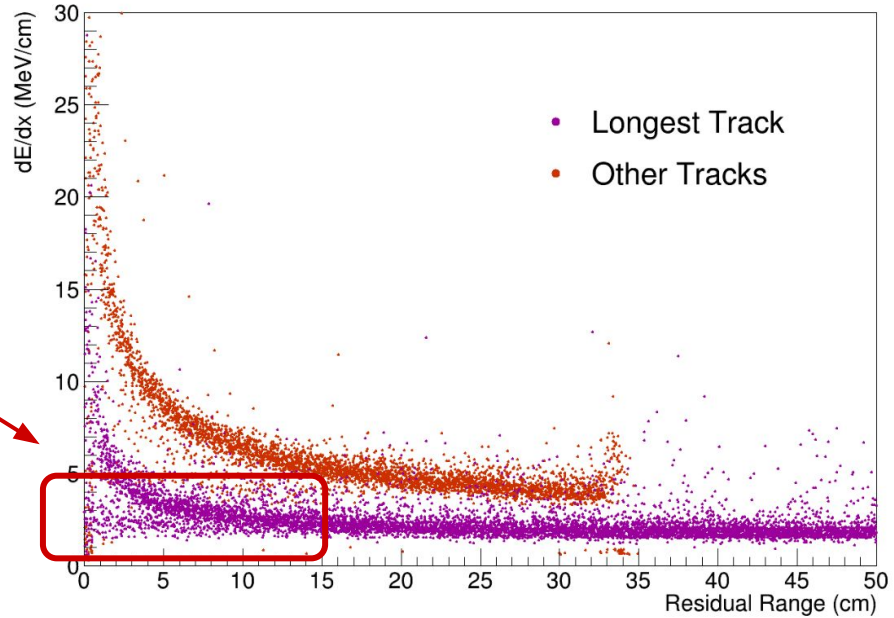




But wait...

What's happening here!?

HINT: Perhaps we're not measuring the residual range from the correct location, but why would that be?





9. Truth Matching

Truth Matching

- When working with simulated files, we know the ‘truth’ of our events i.e. what the generator actually produced
- We can match (or backtrack) the reconstructed PFParticles to the simulated MCParticles
- Let’s recreate our dE/dx plot, but this time using true information to identify muons and protons

TruthMatchUtils

- Our very own Dominic Brailsford created some ‘backtracking’ tools that can be found here: https://github.com/LArSoft/larsim/blob/develop/larsim/Utils/TruthMatchUtils.h

- We’re going to use the function below to get the ID of the MCParticle that matches the input hit collection (from the PFParticle)

```
// Get the g4id of the MCParticle that 'owns' the most hits in the track
int g4id = TruthMatchUtils::TrueParticleIDFromTotalRecoHits(clockData, track_hits, 1);
```

1 indicates that we want to ‘roll up’ shower hierarchies to the leading electron/photon

- You’ll see that we need:
  1. the track’s recob::Hits (use track-hit associations like you’ve done before)
  2. the clockData which we obtain from the 'DetectorClocksService'

TruthMatchUtils

1. We'll get the track → recob::Hits association in the usual way

```
art::FindManyP<recob::Hit> trackHitAssoc(trackHandle, e, fTrackLabel);
```

```
std::vector<art::Ptr<recob::Hit>> track_hits = trackHitAssoc.at(track.key());
```

In our 'tree filling' loop

2. The clockData is obtained from the 'DetectorClocksService', like so:

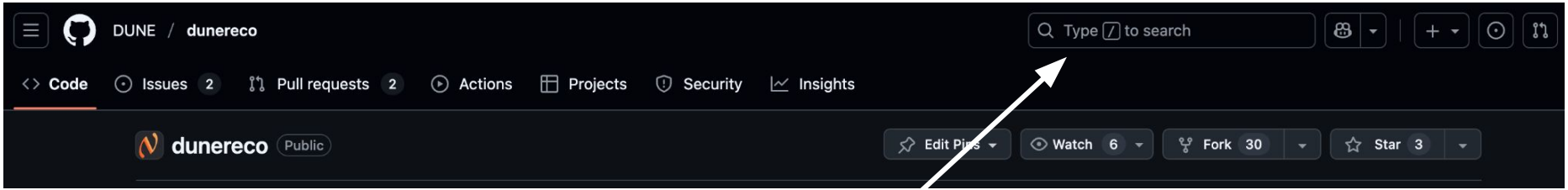
```
auto const clockData = art::ServiceHandle<detinfo::DetectorClocksService>()->DataFor(e);
```

Put this at the top of the analyze function

GitHub searching...

- This function gives the g4id (the trackID) of the matched MCParticle
- We need to use the ParticleInventoryService to get the MCParticle
- I don't want to give you the function here.. so why don't you search a relevant GitHub repo (e.g. dunereco) for examples to see how this is used

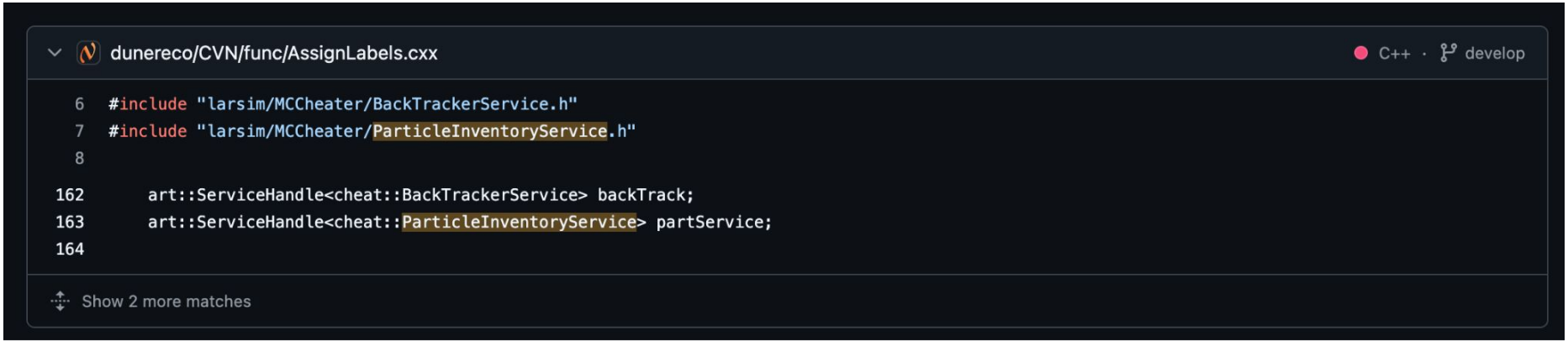
I think that you have to be signed into GitHub for this to work



Search ParticleInventoryService here

GitHub searching…

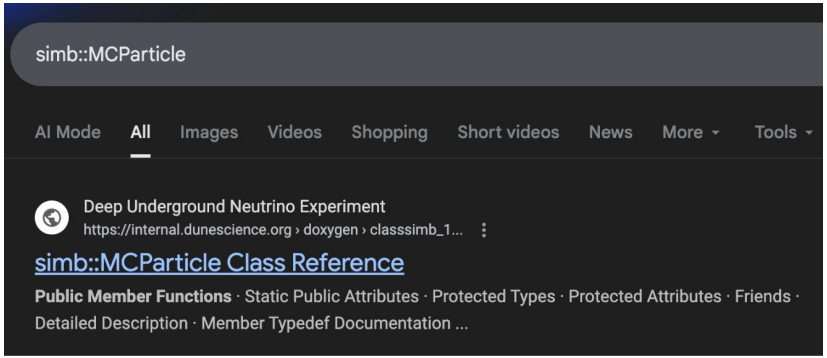
- Scroll through the options until you find something that might be promising, e.g.:

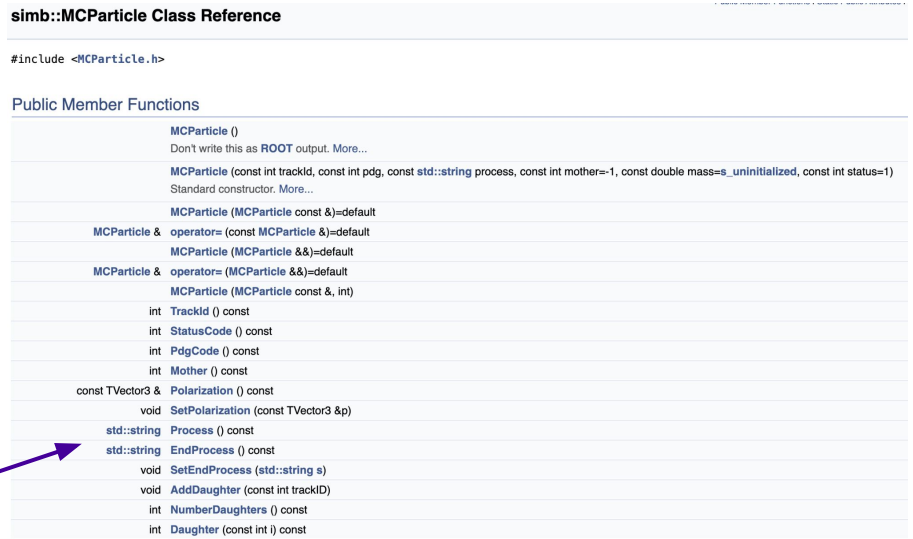


- You'll only be shown the area around the first mention of your search keyword, open up the file to find the ParticleInventoryService function that gives the MCParticle from the TrackID

MCParticle

- So now you have the MCParticle, but how do we get the PDG code?
- Google the class name simb::MCParticle to find the doxygen overview of the MCParticle class methods and member variables.





simb::MCParticle Class Reference

`#include <MCParticle.h>`

Public Member Functions

	MCParticle () Don't write this as **ROOT** output. More...
	MCParticle (const int trackId, const int pdg, const **std::string** process, const int mother=-1, const double mass=**s_uninitialized**, const int status=1) Standard constructor. More...
	MCParticle (**MCParticle** const &)=default
MCParticle &	**operator=** (const **MCParticle** &)=default
	MCParticle (**MCParticle** &&)=default
MCParticle &	**operator=** (**MCParticle** &&)=default
	MCParticle (**MCParticle** const &, int)
int	**TrackId** () const
int	**StatusCode** () const
int	**PdgCode** () const
int	**Mother** () const
const TVector3 &	**Polarization** () const
void	**SetPolarization** (const TVector3 &p)
std::string	**Process** () const
std::string	**EndProcess** () const
void	**SetEndProcess** (**std::string** s)
void	**AddDaughter** (const int trackID)
int	**NumberDaughters** () const
int	**Daughter** (const int i) const

- See if you can find the function that gives the MCParticle's PDG code

muon = 13 proton = 2212

More at: https://pdg.lbl.gov/2007/reviews/montecarlorpp.pdf

Writing out the BacktrackedPDG

- This should be very familiar
  1. Create a fBacktrackedPDG vector (of integers) to store this new information
  2. Create a new branch on the tree, and link it to the fBacktrackedPDG vector
  3. Make sure that it is reset for each event
  4. Add the backtracked PDG for each reconstructed PFParticle

Header Errors

- Attempt to build. You're going to run into some errors (teehee)
- I've done this on purpose, because I think that they represent the most common build errors you'll come across when writing an analyser

```
/home/lar25-t09/sbndcode_2025/srcs/sbndcode/sbndcode/Workshop/Analysis/AnalyseEvents_module.cc:244:49: error: 'detinfo' was not declared in t
his scope
  244 |       auto const clockData = art::ServiceHandle<detinfo::DetectorClocksService>()->DataFor(e);
      |                                                 ^~~~~~~
/home/lar25-t09/sbndcode_2025/srcs/sbndcode/sbndcode/Workshop/Analysis/AnalyseEvents_module.cc:244:79: error: template argument 1 is invalid
  244 |       auto const clockData = art::ServiceHandle<detinfo::DetectorClocksService>()->DataFor(e);
      |                                                                               ^
/home/lar25-t09/sbndcode_2025/srcs/sbndcode/sbndcode/Workshop/Analysis/AnalyseEvents_module.cc:244:79: error: template argument 2 is invalid
/home/lar25-t09/sbndcode_2025/srcs/sbndcode/sbndcode/Workshop/Analysis/AnalyseEvents_module.cc:246:35: error: 'Hit' is not a member of 'recob
'
  246 |       std::vector<art::Ptr<recob::Hit>> track_hits = trackHitAssoc.at(track.key());
      |                                   ^~~
/home/lar25-t09/sbndcode_2025/srcs/sbndcode/sbndcode/Workshop/Analysis/AnalyseEvents_module.cc:246:35: error: template argument 1 is invalid
```

We're missing some header files!

- I normally find the headers I need by looking at the usage in other files, so use your new GitHub search skills to find out the needed headers

CMakeList Errors

```
3.10-2.17-e26-prof/lib:/cvmfs/larsoft.opensciencegrid.org/products/cetlib/v3_18_02/slf7.x86_64.e26.prof/lib && :
/usr/bin/ld: Dwarf Error: found dwarf version '5', this reader only handles version 2, 3 and 4 information.
sbndcode/sbndcode/Workshop/Analysis/CMakeFiles/sbndcode_Workshop_Analysis_AnalyseEvents_module.dir/AnalyseEvents_module.cc.o: In function `te
st::AnalyseEvents::analyze(art::Event const&)':
AnalyseEvents_module.cc:(.text+0x37e3): undefined reference to `TruthMatchUtils::TrueParticleIDFromTotalRecoHits(detinfo::DetectorClocksData
const&, std::vector<art::Ptr<recob::Hit>, std::allocator<art::Ptr<recob::Hit> > > const&, bool)'
AnalyseEvents_module.cc:(.text+0x3863): undefined reference to `TruthMatchUtils::Valid(int)'
collect2: error: ld returned 1 exit status
ninja: build stopped: subcommand failed.


FATAL ERROR: stage install FAILED for MRB project larsoft v10_06_00 with code 1
```

Our build is missing some libraries!

- You'll find a CMakeLists.txt file in the Analysis direction, which tells the build the libraries that it needs to be aware of when building the analyser
- We need to make it aware of the ParticleInventoryService and TruthUtils libraries, add to the file:

```
larsim::MCCheater_ParticleInventoryService_service
larsim::Utils
```

Runtime Errors

```
---- ServiceNotFound BEGIN
  Unable to create ServiceHandle.
  Perhaps the FHiCL configuration does not specify the necessary service?
  The class of the service is noted below...
  ---- ServiceNotFound BEGIN
    ServicesManager unable to find the service of type 'cheat::BackTrackerService'.
  ---- ServiceNotFound END
```

- Your code should now build, but you might see the above runtime error
- If your code doesn't complain, your 'BacktrackedPDG' branch will likely be filled by -1
- We need to 'configure' the BackTrackerService and ParticleInventoryService in the 'services' section of the run fcl:

```
ParticleInventoryService:  @local::sbnd_particleinventoryservice
BackTrackerService:        @local::sbnd_backtrackerservice
DetectorClocksService:     @local::sbnd_detectorclocks
```

Using the Truth (1)

- Let's again make two dE/dx plots, this time splitting the entries by the Backtracked PDG

```
root[0] ana->cd()
```

```
root[1] TH2D *hMuon = new TH2D("hMuon","dE/dx vs. Residual Range", 200, 0, 50, 200, 0, 30)
```

```
root[2] TH2D *hProton = new TH2D("hProton","dE/dx vs. Residual Range", 200, 0, 50, 200, 0, 30)
```

Applying our PID (2)

This time we need to include our condition on the draw command…

```
root[3] tree->Draw("childTrackdEdx:childTrackResRange>>hMuon", "backtrackedPDG == 13", "")
```

```
root[4] tree->Draw("childTrackdEdx:childTrackResRange>>hProton", "backtrackedPDG == 2212", "same")
```

We need to tell the two apart… Let's draw them in different colours!

```
root[5] hMuon->SetMarkerColor(kGreen+3)
```

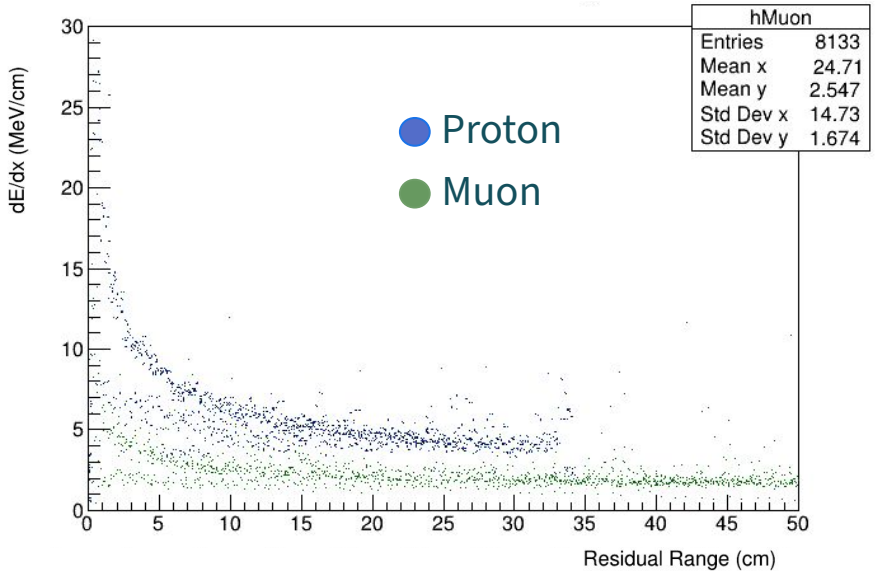
Alternative colour options are here: https://root.cern.ch/doc/master/classTColor.html

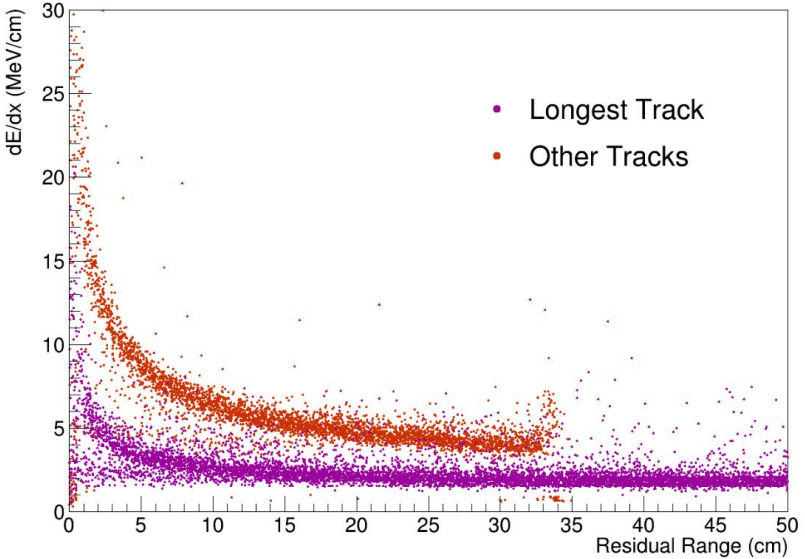
```
root[6] hProton->SetMarkerColor(kAzure-1)
```

```
root[6] c1->Modified()
```

Tell the canvas (default c1) to implement these changes and redraw the canvas

Now remake your track length plot!





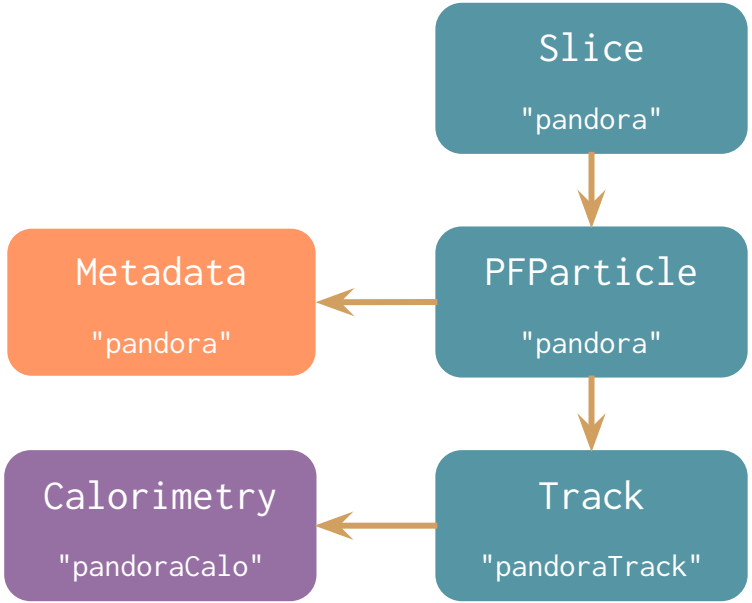
- The truth-based selection and longest-track-based selections agree!
- The longest-track assumption isn't too bad!



10. Recovering t_0

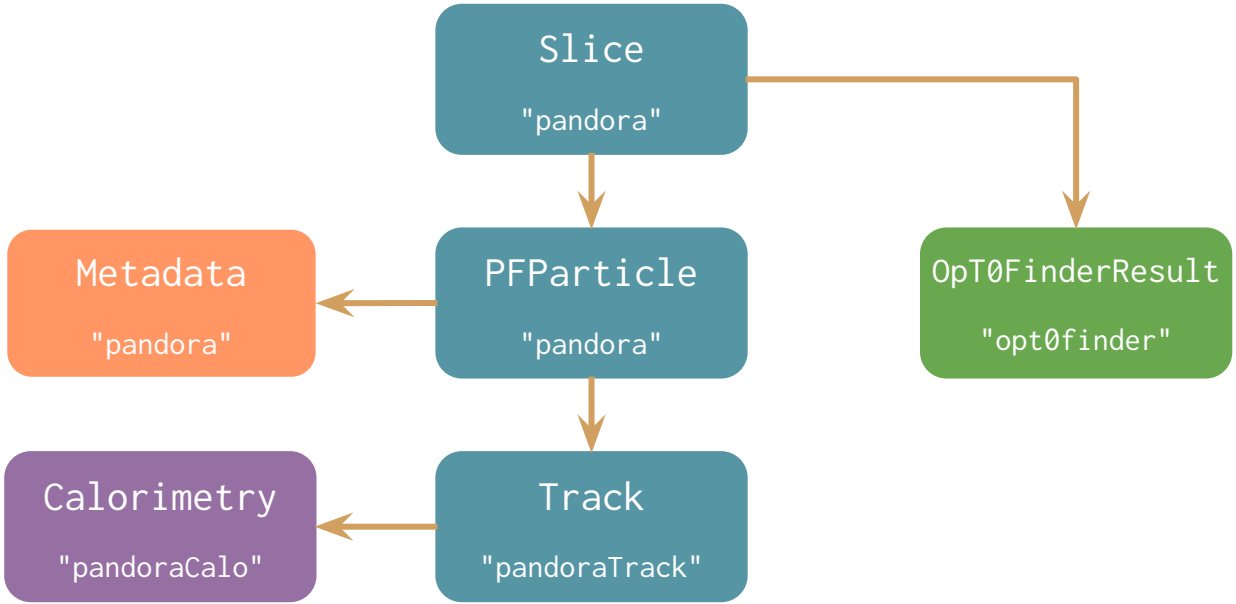
Detector system associations

- We have previously looked at associations between reconstructed quantities for the purpose of accessing geometry and calorimetry information about the particles in our events



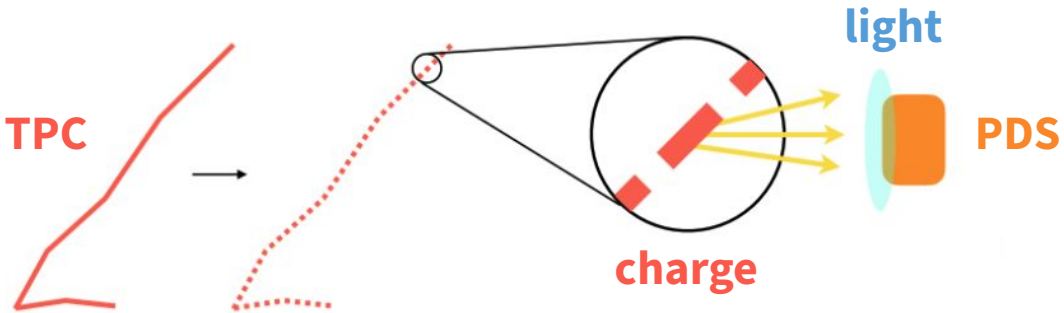
Detector system associations

- But we can also look at the the associations that link different detector systems: TPC (charge), PDS (light), and CRT (outer cosmic-ray tagger)



What is this t0?

- The t0 is the time at which we think the interaction happened
- We obtain this from the photon-detection system (PDS) and use it to set the timing (x-position) of the TPC charge-reconstructed slice
- This is done by ‘charge-light matching’ where we try to match the **charge** image we saw in the TPC to the **light** image we saw in the PDS
- The t0 can be used to separate interactions that coincide with the beam’s arrival (neutrinos) with those that don’t (cosmic-rays)



Adding Flash Matching Information

We're going to leave you to try and add this one on your own. The object is called `sbn::OpT0Finder` and lives here. You will need to:

1. Add the relevant header
2. Add the module label to the fcl file and access it in the analyzer
3. Use the association to access the object
4. Sometimes there are multiple OpT0Finder results per slice, you should pick the one with the largest score variable.
5. Save the time variable from the object to your tree.

Give this a go, looking back on the previous exercises for help. If you get stuck, the solutions are in the following slides.

Adding OpT0Finder

1. Add the relevant header

```
// SBN(D) includes
#include "sbnobj/Common/Reco/OpT0FinderResult.h"
```

2. Add the module label to the fcl file and access it in the analyzer

```
OpT0FinderLabel:  "opt0finder"
```

```
std::string fOpT0FinderLabel;
```

```
fOpT0FinderLabel(p.get<std::string>("OpT0FinderLabel"))
```

3. Use the association to access the object

```
 art::FindManyP<sbn::OpT0Finder> sliceOpT0Assoc(sliceHandle, e, fOpT0FinderLabel);
```

Accessing OpT0Finder

- A slice sometimes has multiple OpT0Finder results, pick the one with the largest score variable
- Save the time variable from the object to your tree

I put this under the if (nuID < 0) check

```
// Get any OpT0Finder results associated with our slice
std::vector<art::Ptr<sbn::OpT0Finder>> opT0s = sliceOpT0Assoc.at(nuSliceKey);

// Occasionally there may be multiple results, let's use the one with the best score
std::sort(opT0s.begin(), opT0s.end(),
        [](auto const& a, auto const& b)
        { return a->score > b->score; });

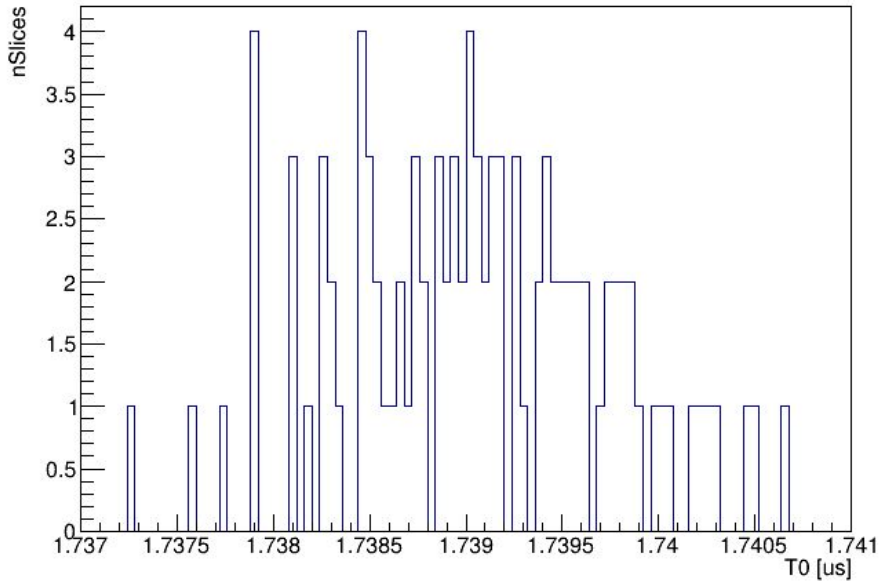
// The best score will now be at the front of the vector (if there were any)
if (opT0s.size() != 0)
fOpT0 = opT0s[0]->time;
```

Here the opT0s are sorted in order of score

You'll need to define fOpT0, add it as a branch to the tree and make sure that it is reset with every event

T0 Results

Remember way back in the simulation tutorial? You defined t0 to be 1600ns.



- Your OpT0 results should give you values close to that original simulated time
- Last year we discovered this number to be off and it took us a long time and asking other experts to understand why
- Worth remembering that all of us still have to ask questions all the time, so never worry about reaching out with questions!

Final notes

ROOT Workflows

- These tutorials focus on using ROOT via a VNC connection
- Trying to open root files (or any visualisation) via a standard ssh connection will result in bad times
- You can often set up a VNC over an ssh connection (e.g. to the Fermilab GPVMs)
- You can also copy root files to your local machine and run root macros locally (the TTree files are much smaller than the art files and root can be compiled on a laptop fairly easily with minimal dependencies)

Some important file locations

Our version of the code lives here:

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis/.FinishedModule/AnalyseEvents_module.cc
```

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis/.FinishedModule/analysisConfig.fcl
```

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis/.FinishedModule/run_analyseEvents.fcl
```

Type `ls -a` *in the directories to see hidden files and directories*

Documentation and additional information

The documentation for each art object/tool we have looked at lives here:

- recob::PFParticle - https://code-doc.larsoft.org/docs/latest/html/classrecob_1_1PFParticle.html
- art::FindManyP - https://code-doc.larsoft.org/docs/latest/html/classart_1_1FindManyP.html
- recob::Track - https://code-doc.larsoft.org/docs/latest/html/classrecob_1_1Track.html
- larpandoraobj::PFParticleMetadata - https://code-doc.larsoft.org/docs/latest/html/classlarpandoraobj_1_1PFParticleMetadata.html
- anab::Calorimetry - https://code-doc.larsoft.org/docs/latest/html/classanab_1_1Calorimetry.html

Remember you can look at all of the objects and their corresponding producers in any reco file by looking at an event dump:

```
lar -c eventdump.fcl -s /path/to/reco/file.root -n 1
```

Some useful doxygen/github

LArSoft-y things:

Doxygen:
https://code-doc.larsoft.org/docs/latest/html/

Github:
https://github.com/LArSoft/

Pandora Github:

https://github.com/PandoraPFA

Experiment-based:

SBN-wide Doxygen:
https://sbnsoftware.github.io/doxygen/

sbndcode Github:
https://github.com/SBNSoftware/sbndcode

MicroBooNE Github:
https://github.com/uboone

DUNE Github:
https://github.com/DUNE

Previous tutorials

Isobel Mawby & Alex Wilkinson's tutorial from 2025 is here:
https://indico.ph.ed.ac.uk/event/381/contributions/4035/

Isobel Mawby & Alex Wilkinson's tutorial from 2025 (DUNE-focused workshop at CERN) is here:
https://indico.cern.ch/event/1461779/contributions/6319649/ (password: LArSAW2025)

Henry Lay & Lan Nguyen's tutorial from 2024 is here:
https://indico.ph.ed.ac.uk/event/313/contributions/3414/

Isobel Mawby & Henry Lay's tutorial from 2023 is here:
https://indico.ph.ed.ac.uk/event/268/contributions/2731/

Ed Tyley & Rhiannon Jones' tutorial from 2022 is here:
https://indico.ph.ed.ac.uk/event/130/contributions/1747/

Ed Tyley & Rhiannon Jones' tutorial from 2021 is here:
https://indico.ph.ed.ac.uk/event/91/contributions/1417/

Owen Goodwin's tutorial from 2020 is here:
https://indico.hep.manchester.ac.uk/static/5856/contributions/12-4-0-slides

Rhiannon Jones' tutorial from 2019 is here: (this link no longer works - SAD.)
https://indico.hep.manchester.ac.uk/getFile.py/access?contribId=13&sessionId=4&resId=0&materialId=slides&confId=5544

Leigh Whitehead's tutorial from 2018 is here: (this link no longer works - SAD.)
https://indico.hep.manchester.ac.uk/getFile.py/access?contribId=13&sessionId=2&resId=0&materialId=slides&confId=5372