The University Of Sheffield.

DEEP UNDERGROUND NEUTRINO EXPERIMENT

Writing your first analyzer

9th September 2022

Latin America-UK LArsoft workshop

Rhiannon Jones
r.s.jones@sheffield.ac.uk
#larsoft_analyser

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

Side note

- I have included what I think will 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

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

The University Of Sheffield.

DEEP UNDERGROUND NEUTRINO EXPERIMENT

The empty **‘analyzer’**

Before we begin

Yesterday, you may have found it very annoying to have to re-run the setup scripts every time you open a terminal. My suggestion is that everyone makes a script that simplifies the process into a single command, as follows.

Once you have opened a terminal, open a file called setup.sh
Put the following lines into that file and save it:

Substitute **dune_workdir** with the name you gave the directory which contains your local products

```
source /cvmfs/dune.opensciencegrid.org/products/dune/setup_dune.sh
cd ~/dune_workdir/
source localProducts_larsoft_v09_56_00d00_prof_e20/setup
mrbsetenv
mrbslp
```

You may need to modify a few elements to match your personal setup, but I'll come round and help you suss it out now.

Before we begin

Once you have made setup.sh, open ~/.bashrc
Add this line:

```
alias setuplarsoft=‘source ~/setup.sh’
```

Close the terminal, open a new one and type:

```
setuplarsoft
```

Let me know what errors you get and we can clean it up.

Once that works, you only need to type setuplarsoft every time you open a terminal, rather than running each line individually.

Initial navigation

Once you're setup, navigate here:

```
cd $MRB_SOURCE/duneana/duneana/Workshop/Analysis
```

there should be a `CMakeLists.txt` and a `build.sh` file.

I have been updating the contents of this directory so you will need to pull any changes from git:

```
git pull
```

If you have any issues after running this command, please let me know!

The skeleton analysis module

There are 2 ways of getting your skeleton analyzer

1. Using a command like this:

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

We will use this next: 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

The skeleton analysis module

There are 2 ways of gettin

1. Using a command like t

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
```

for starting somethi

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

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

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

The skeleton analysis module

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

```
source /cvmfs/dune.opensciencegrid.org/products/dune/setup_dune.sh
source $MRB_TOP/localProducts_larsoft_v09_56_00_e20_prof/setup
mrbslp
```

1. Navigate here:

```
cd $MRB_SOURCE/duneana/duneana/Workshop/Analysis
```

2. Type this command:

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

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

What do we have so far?

You should now have a file called **`AnalyzeEvents_module.cc`** and the **`CMakeLists.txt`** in your directory

Open your analyzer module!

The top section should look something like the snippet on the right

```
////////////////////////////////////////////////////////////////////////
// Class:       AnalyzeEvents
// Plugin Type: analyzer (Unknown Unknown)
// File:        AnalyzeEvents_module.cc
//
// Generated at Tue Sep  6 16:33:28 2022 by dune28 using cetskelgen
// from  version .
////////////////////////////////////////////////////////////////////////

#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 AnalyzeEvents;
}


class test::AnalyzeEvents : public art::EDAnalyzer {
public:
  explicit AnalyzeEvents(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.
  AnalyzeEvents(AnalyzeEvents const&) = delete;
  AnalyzeEvents(AnalyzeEvents&&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents const&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents&&) = delete;

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

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

private:

  // Declare member data here.

};
```


12

What do we have so far?

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

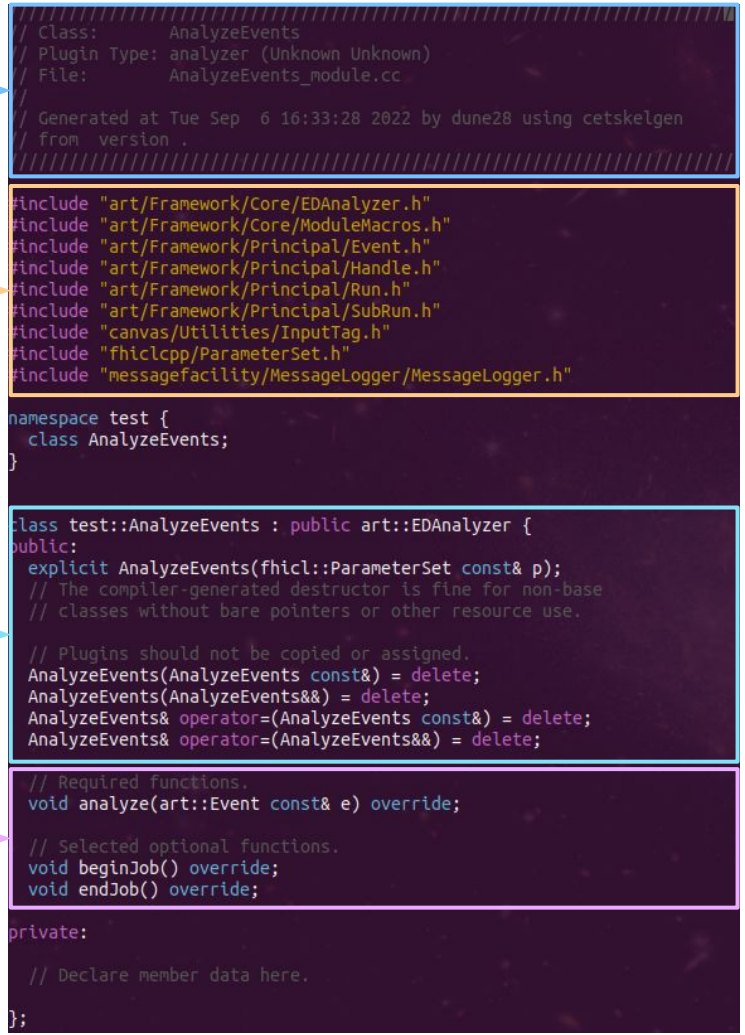
These are the default headers which should hopefully allow the empty analyzer to build

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:       AnalyzeEvents
// Plugin Type: analyzer (Unknown Unknown)
// File:        AnalyzeEvents_module.cc
//
// Generated at Tue Sep  6 16:33:28 2022 by dune28 using cetskelgen
// from  version .
////////////////////////////////////////////////////////////////////////

#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 AnalyzeEvents;
}


class test::AnalyzeEvents : public art::EDAnalyzer {
public:
  explicit AnalyzeEvents(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.
  AnalyzeEvents(AnalyzeEvents const&) = delete;
  AnalyzeEvents(AnalyzeEvents&&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents const&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents&&) = delete;

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

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

private:

  // Declare member data here.

};
```


13

What do we have so far?

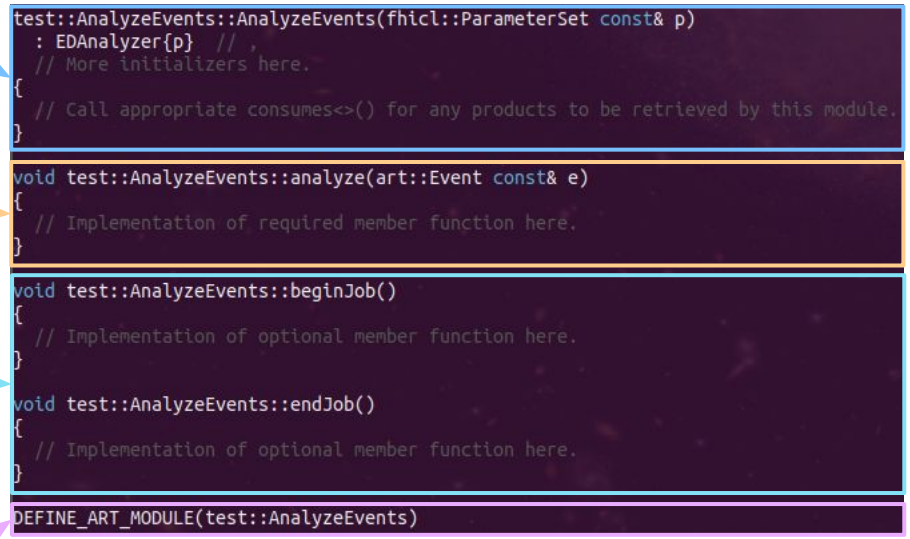
Scroll down to the next chunk of code in your analyzer module

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

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

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

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



You should now have reached the end of the file

14

Adding an output (T)Tree → Compiling and running the code

Adding an output tree

We will be modifying various elements of the code before compiling

Add relevant LArSoft & ROOT headers

Declare TTree and event-based variables

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

Create your TTree & add branches for the variables we want to fill

Note: The order follows how the file reads

```
#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"

// Additional framework includes
#include "art_root_io/TFileService.h"

// ROOT includes
#include <TTree.h>
```

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

  // Tree variables
  unsigned int fEventID;
};
```

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

  // Store the outputs in the TTree
  fTree->Fill();
}
```

```
void test::AnalyzeEvents::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 the 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 analysis (An include fcl)
  - This is where we point the analyzer to the objects/parameters we want to access from the input files
- The second will be used to run our analysis (A run/job fcl)
  - This links together the configuration file and the analysis module

---

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

Your chosen name for this parameter set

See what this does (and more best practices) here

```
BEGIN_PROLOG

analyzeEvents:
{
  module_type:          "AnalyzeEvents"
}

END_PROLOG
```

Links the fhicl file to the analysis module using the name you gave your analyzer class

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

Open up another file, e.g.
`run_analyzeEvents.fcl`
& fill it with this:

```
#include "analysisConfig.fcl"
#include "services_dune.fcl"

process_name: AnalyzeEvents # 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::dunefd_1x2x6_simulation_services
}

physics:
{
  analyzers:
  {
    ana: @local::analyzeEvents  #inserts into workflow, matches name in config fcl
  }
  path0:        [ ana ]
  end_paths:    [ path0 ]
}
```

The University Of Sheffield.

DEEP UNDERGROUND NEUTRINO EXPERIMENT

Let's try running it

Pre-made reconstructed events

Don't panic!

The location of the pre-made reconstruction file is:

```
/home/share/september2022/reconstruction/reco_1mu1p.root
```

Compiling and running your code

First, compile what you've written so far

From the `$MRB_SOURCE/duneana/duneana/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_analyzeEvents.fcl -s /path/to/input/file.root -n 10
```

Let's see what we've got in the output file...

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

Compiling and running your code

First, compile what you've written so far

From the `$MRB_SOURCE/duneana/duneana/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_analyzeEvents.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

Let's see what we've got in the output file...

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

Looking at the output in ROOT

```
root -l analysisOutput.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*

Here you can see the output (T)Tree that we created

Your **`tree`** exists and contains the **`eventIDs`**! Success! (hopefully)

```
root [0]
Attaching file analysisOutput.root as _file0...
(TFile *) 0x214e200
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 *
************************
*        0 *         1 *
*        1 *         2 *
*        2 *         3 *
*        3 *         4 *
*        4 *         5 *
*        5 *         6 *
*        6 *         7 *
*        7 *         8 *
*        8 *         9 *
*        9 *        10 *
************************
(long long) 10
```

Accessing `PFParticles` and adding them to the output tree

Accessing the PFParticles

Add the new headers we need

```
// Additional LArSoft includes
#include "lardataobj/RecoBase/PFParticle.h"

// ROOT includes
#include <TTree.h>

// STL includes
#include <string>
#include <vector>
```

We will discuss in detail how to implement this in the `analyze` function next!

Some new parameters to add to our TTree
Including the label for the **`PFParticle`** module

```
  // Tree variables
  unsigned int fEventID;
  unsigned int fNPFParticles;
  unsigned int fNPrimaries;
  int fNPrimaryDaughters;

  // Define input labels
  const std::string fPFParticleLabel;
```

This links to your configuration fcl.
We'll look at how later.

In the class constructor, extract the label for the **`PFParticle`** producer (**pandora**) from our configuration fhicl

```
test::AnalyzeEvents::AnalyzeEvents(fhicl::ParameterSet const& p)
  : EDAnalyzer{p},
  fPFParticleLabel(p.get<std::string>("PFParticleLabel"))
{
  // Call appropriate consumes<>() for any products to be retrieved by this module.
}
```

Define the new branches in the TTree

```
  // Add branches to the TTree
  fTree->Branch("eventID", &fEventID);
  fTree->Branch("nPFParticles",&fNPFParticles);
  fTree->Branch("nPrimaries",&fNPrimaries);
  fTree->Branch("nPrimaryDaughters",&fNPrimaryDaughters);
```

Accessing the PFParticles

We're now inside your `analyze` function

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

// Set all counters to 0 for the current event
fNPFParticles      = 0;
fNPrimaries        = 0;
fNPrimaryDaughters = 0;
```

Empty the counters at the start of the event

Accessing the PFParticles

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

// Set all counters to 0 for the current event
fNPFParticles      = 0;
fNPrimaries        = 0;
fNPrimaryDaughters = 0;

// Load the PFParticles from Pandora
art::Handle<std::vector<recob::PFParticle>> pfpHandle;
std::vector<art::Ptr<recob::PFParticle>>    pfpVec;
if(e.getByLabel(fPFParticleLabel, pfpHandle))
  art::fill_ptr_vector(pfpVec, pfpHandle);

// If there are no PFParticles then skip the event
if(pfpVec.empty())
  return;
```

The analysis objects are always formatted such that we access them from a vector. The **`art::Handle< std::vector< … > >`** is the art wrapper which holds each vector.

In our case, we want the **`PFParticles`** from the RecoBase, **`recob`**, using the appropriate module label: **`pandora`**.

We then make sure the **`art::Handle`** is valid before filling the vector of objects to analyze.

Accessing the PFParticles

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

// Set all counters to 0 for the current event
fNPFParticles      = 0;
fNPrimaries        = 0;
fNPrimaryDaughters = 0;

// Load the PFParticles from Pandora
art::Handle<std::vector<recob::PFParticle
std::vector<art::Ptr<recob::PFParticle>>
if(e.getByLabel(fPFParticleLabel, pfpHand
  art::fill_ptr_vector(pfpVec, pfpHandle)

// If there are no PFParticles then skip
if(pfpVec.empty())
  return;

// Initialise the neutrino ID
size_t neutrinoID(std::numeric_limits<size_t>::max());

// Loop over the PFParticles and find the neutrino
for(const art::Ptr<recob::PFParticle> &pfp: pfpVec){
  fNPFParticles++;

  // Check that we are looking at a primary particle with a neutrino pdg code. If not, skip the PFParticle
  if(!(pfp->IsPrimary() && (std::abs(pfp->PdgCode()) == 14 || std::abs(pfp->PdgCode()) == 12)))
    continue;
  fNPrimaries++;

  neutrinoID         = pfp->Self();
  fNPrimaryDaughters = pfp->NumDaughters();
} // PFParticle loop

// Check that we found a neutrino
if(neutrinoID == std::numeric_limits<size_t>::max())
  return;
```

We can now loop over the **`PFParticles`** for events in which they have been produced.
`IsPrimary()` returns true for the reconstructed neutrino. **`NumDaughters()`** used here tells us the number of daughters the neutrino **`PFParticle`** has.
To check the available functions within a class you can check the doxygen: https://nusoft.fnal.gov/larsoft/doxsvn/html/classrecob_1_1PFParticle.html

Accessing the PFParticles

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

  // Set all counters to 0 for the current event
  fNPFParticles      = 0;
  fNPrimaries        = 0;
  fNPrimaryDaughters = 0;

  // Load the PFParticles from Pandora
  art::Handle<std::vector<recob::PFParticle>> pfpHandle;
  std::vector<art::Ptr<recob::PFParticle>>    pfpVec;
  if(e.getByLabel(fPFParticleLabel, pfpHandle))
    art::fill_ptr_vector(pfpVec, pfpHandle);

  // If there are no PFParticles then skip the event
  if(pfpVec.empty())
    return;

  // Initialise the neutrino ID
  size_t neutrinoID(std::numeric_limits<size_t>::max());

  // Loop over the PFParticles and find the neutrino
  for(const art::Ptr<recob::PFParticle> &pfp: pfpVec){
    fNPFParticles++;

    // Check that we are looking at a primary particle with a neutrino pdg code. If not, skip the PFParticle
    if(!(pfp->IsPrimary() && (std::abs(pfp->PdgCode()) == 14 || std::abs(pfp->PdgCode()) == 12)))
      continue;
    fNPrimaries++;

    neutrinoID         = pfp->Self();
    fNPrimaryDaughters = pfp->NumDaughters();
  } // PFParticle loop

  // Check that we found a neutrino
  if(neutrinoID == std::numeric_limits<size_t>::max())
    return;

  // Store the outputs in the TTree
  fTree->Fill();
}
```

The entire code-block for this section of the `analyze` *function*

Fhicl configuration file linking & running

Add the **PFParticle** module label "**pandora**" to the configuration file
Note that the parameter name matches the string we passed to the constructor of the analyzer
Running `eventdump.fcl` prints the products and the modules names

```
analyzeEvents:
{
  module_type:          "AnalyzeEvents"

  PFParticleLabel:      "pandora"
```

```
source build.sh
```

Compile changes

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

Run analyzer

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

Check output

What the output looks like now

Our tree should now have 3 new branches

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

Open the output file

We can check that everything looks sensible:

```
ana->cd
```

Move into the output directory

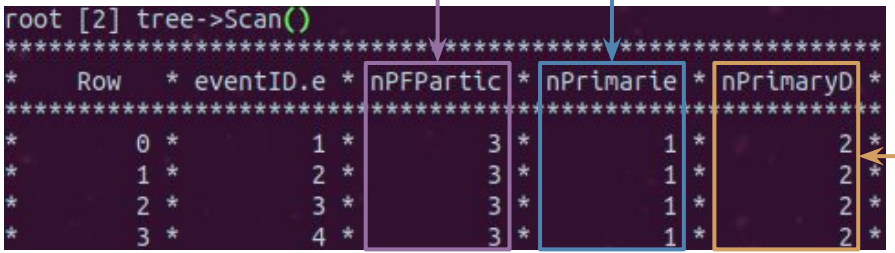
`nPrimaries` should be 0 or 1 in our sample
0 if we didn't reconstruct anything

`nPFParticles` != `nPrimaries` + `nPrimaryDaughters`
As we can have some non-primary particles

nPFParticles tells us how many particle we have reconstructed

nPrimaries is the number of neutrinos

nPrimaryDaughters is the number of primary particles (Daughters of the Neutrinos) we have reconstructed



The University Of Sheffield.

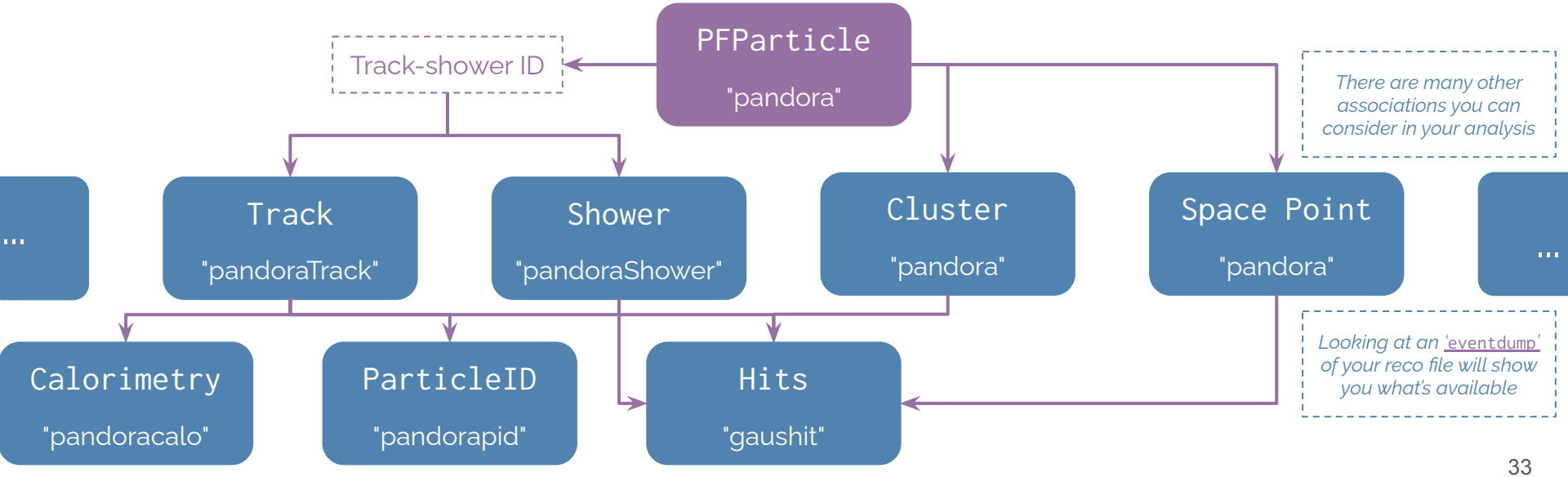
DEEP UNDERGROUND NEUTRINO EXPERIMENT

Associations

What is an association?

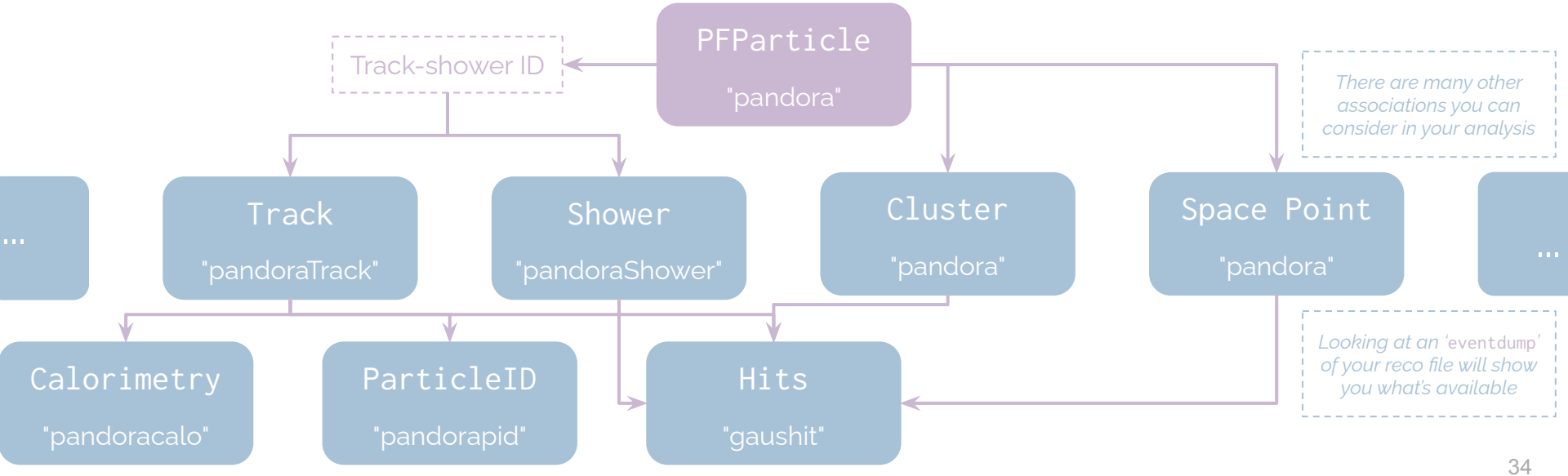
LArSoft uses associations to make links between different objects

- The `recob::PFParticles` have associations to other objects
- Below is an example of how some are linked to them, producer names are defined in " … "



What is an association?

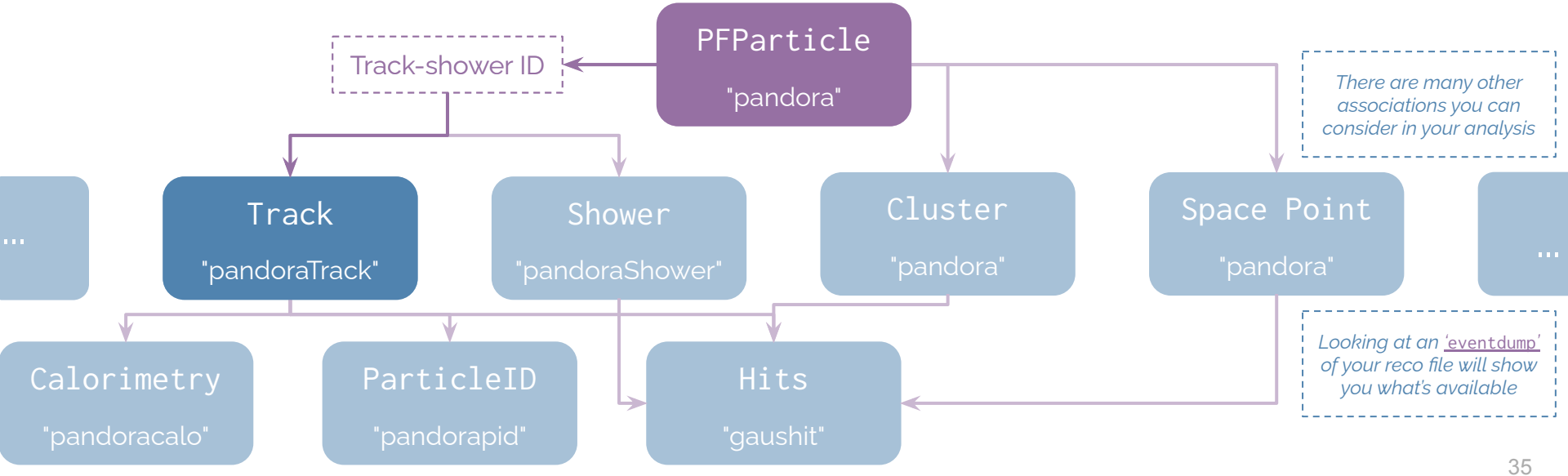
Don't panic! We won't look at all of these.



What is an association?

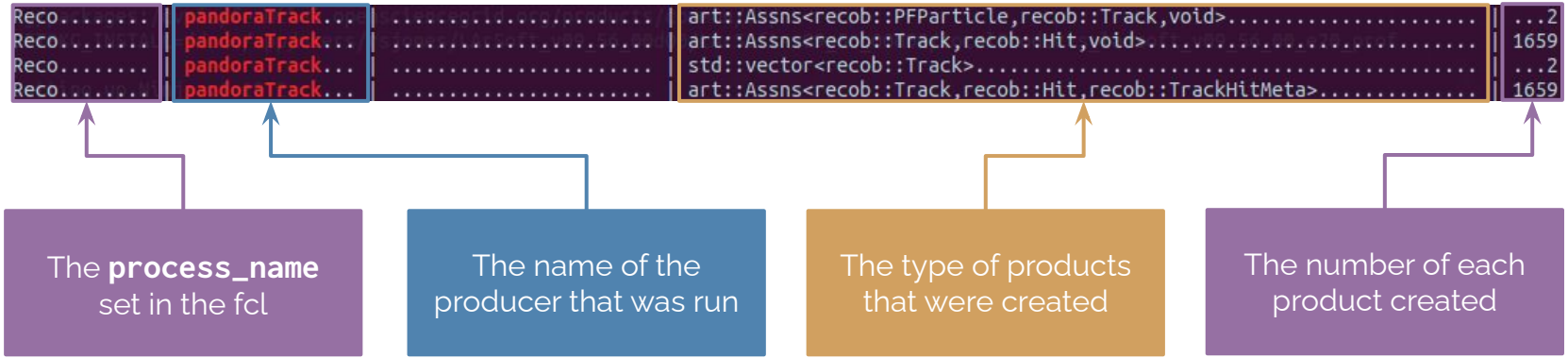
To start with, we'll simply access `recob::Track` associations to `recob::PFParticles`

- *since we are interested in finding a muon and a proton*



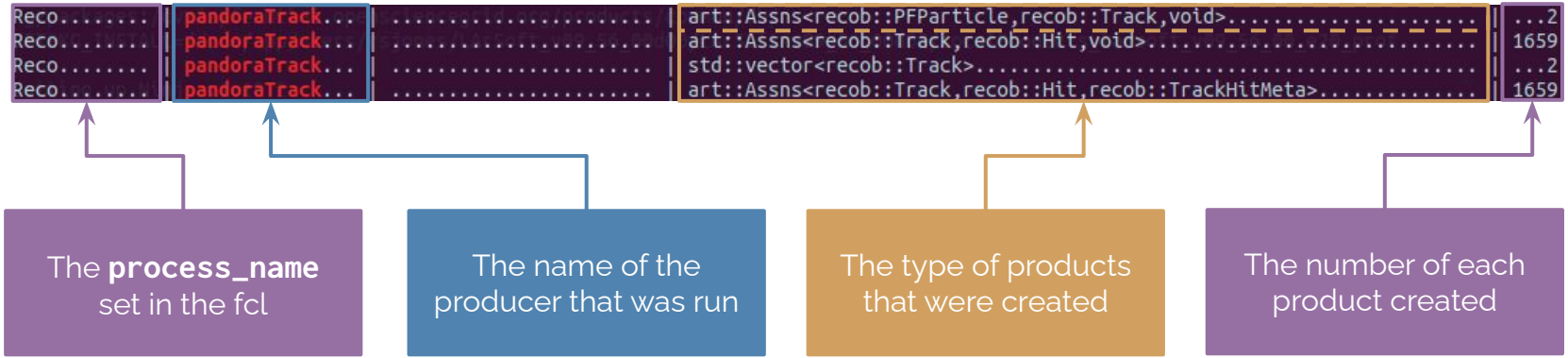
Finding the associations in an event

Running **`eventdump.fcl`** will show us not only the products in the event but the associations between them. Here is everything produced by **`pandoraTrack`**



Finding the associations in an event

Running **eventdump.fcl** will show us not only the products in the event but the associations between them. Here is everything produced by **pandoraTrack**



We want the association between **recob::PFParticle** and **recob::Track**

Now let's apply this to the analysis

```
// Additional framework includes
#include "art_root_io/TFileService.h"
#include "canvas/Persistency/Common/FindManyP.h"

// Additional LArSoft includes
#include "lardataobj/RecoBase/PFParticle.h"
#include "lardataobj/RecoBase/Track.h"
```

These are the additional headers you'll need.
`FindManyP` is the class which 'finds many' pointers to a certain type of object.
In our case, this is used initially as follows:

```
// Get the tracks associated with the PFParticles
art::FindManyP<recob::Track> pfpTrackAssns(pfpVec, e, fTrackLabel);
```

Here we are accessing the `recob::Track` objects associated with everything in the `pfpVec`.
The `recob::Track` objects we want have been produced by the `fTrackLabel` module.
Once again, this will be linked to the configuration file shortly.

The details (bitty part)

In the configuration file add the label of the track producer

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

Add a new field to store the TrackLabel that we set in the fcl above

Initialise the TrackLabel from the configuration

In analysisConfig.fcl

```
module_type:          "AnalyzeEvents"

PFParticleLabel:      "pandora"
TrackLabel:           "pandoraTrack"
```

In analyzeEvents_module.cc

```
  // Tree variables
  unsigned int fEventID;
  unsigned int fNPFParticles;
  unsigned int fNPrimaries;
  int fNPrimaryDaughters;

  std::vector<float> fDaughterTrackLengths;

  // Define input labels
  const std::string fPFParticleLabel;
  const std::string fTrackLabel;
};


test::AnalyzeEvents::AnalyzeEvents(fhicl::ParameterSet const& p)
  : EDAnalyzer{p},
  fPFParticleLabel(p.get<std::string>("PFParticleLabel")),
  fTrackLabel(p.get<std::string>("TrackLabel"))
```

Creating the output

Reset the values stored in the vector for each event

```
// Set all counters to 0 for the current event
fNPFParticles      = 0;
fNPrimaries        = 0;
fNPrimaryDaughters = 0;
fDaughterTrackLengths.clear();                    analyze(…)
```

Add a new branch to the TTree using the vector defined on the previous slide

```
// Add branches to the TTree                      beginJob()
fTree->Branch("eventID", &fEventID);
fTree->Branch("nPFParticles",&fNPFParticles);
fTree->Branch("nPrimaries",&fNPrimaries);
fTree->Branch("nPrimaryDaughters",&fNPrimaryDaughters);
fTree->Branch("daughterTrackLengths",&fDaughterTrackLengths);
```

The details, in **analyze**

This is where you use **FindManyP** (from previous slide)

Checking that the parent of the current **PFParticle** is the neutrino

Defining the vector of **Track** objects associated to the current **PFParticle**
There should be only a single track associated with each **PFParticle**

Filling the vector of **Track** lengths we declared earlier
Done for every **PFParticle** with an associated **Track**

```
  // Check that we found a neutrino
  if(neutrinoID == std::numeric_limits<size_t>::max())
    return;

  // Get the tracks associated with the PFParticles
  art::FindManyP<recob::Track> pfpTrackAssns(pfpVec, e, fTrackLabel);

  for(const art::Ptr<recob::PFParticle> &pfp: pfpVec){
    // We are only interested in the neutrino daughter particles
    if(pfp->Parent() != neutrinoID)
      continue;

    // Get the tracks associated with this PFParticle
    const std::vector<art::Ptr<recob::Track>> pfpTracks(pfpTrackAssns.at(pfp.key()));

    // There should only ever be 0 or 1 tracks associated with a single PFParticle
    if(pfpTracks.size() == 1){
      // Get the first element of the vector
      const art::Ptr<recob::Track> &pfpTrack(pfpTracks.front());

      // Add paramters from the track to the branch vector
      fDaughterTrackLengths.push_back(pfpTrack->Length());
    } // PFParticle Track

  } // PFParticles

  // Store the outputs in the TTree
  fTree->Fill();
}
```

The University Of Sheffield.

DEEP UNDERGROUND NEUTRINO EXPERIMENT

A little more of an in depth analysis of the output

Let's look at the track lengths

Once you have compiled and run your analysis module once more, this time over all your events, open the output file
We'll open a `TBrowser` and have a look at the distribution of track lengths

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

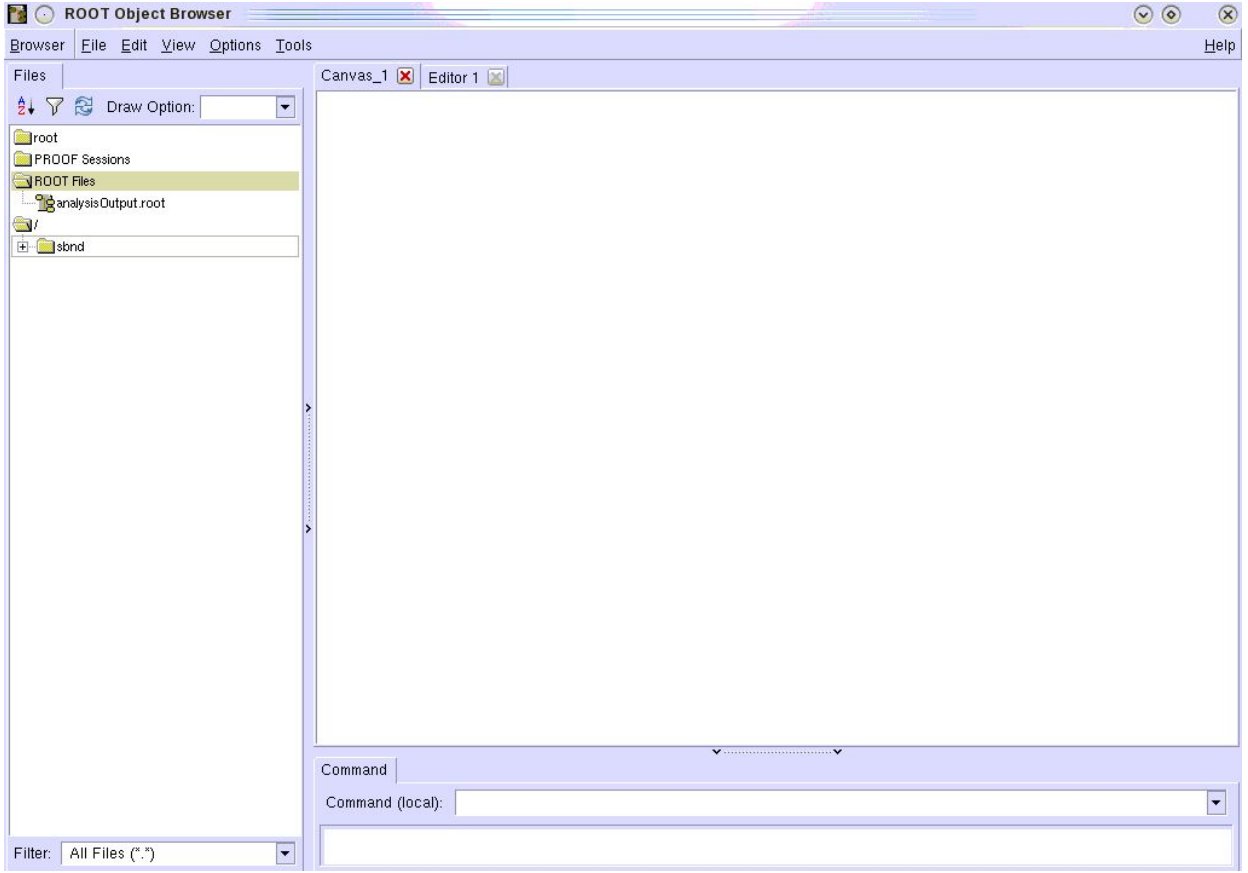
```
lar -c run_analyzeEvents.fcl -s /path/to/input/file.root
```

When you are inside the output file, open up a `TBrowser` like this:

```
root[0] new TBrowser
```

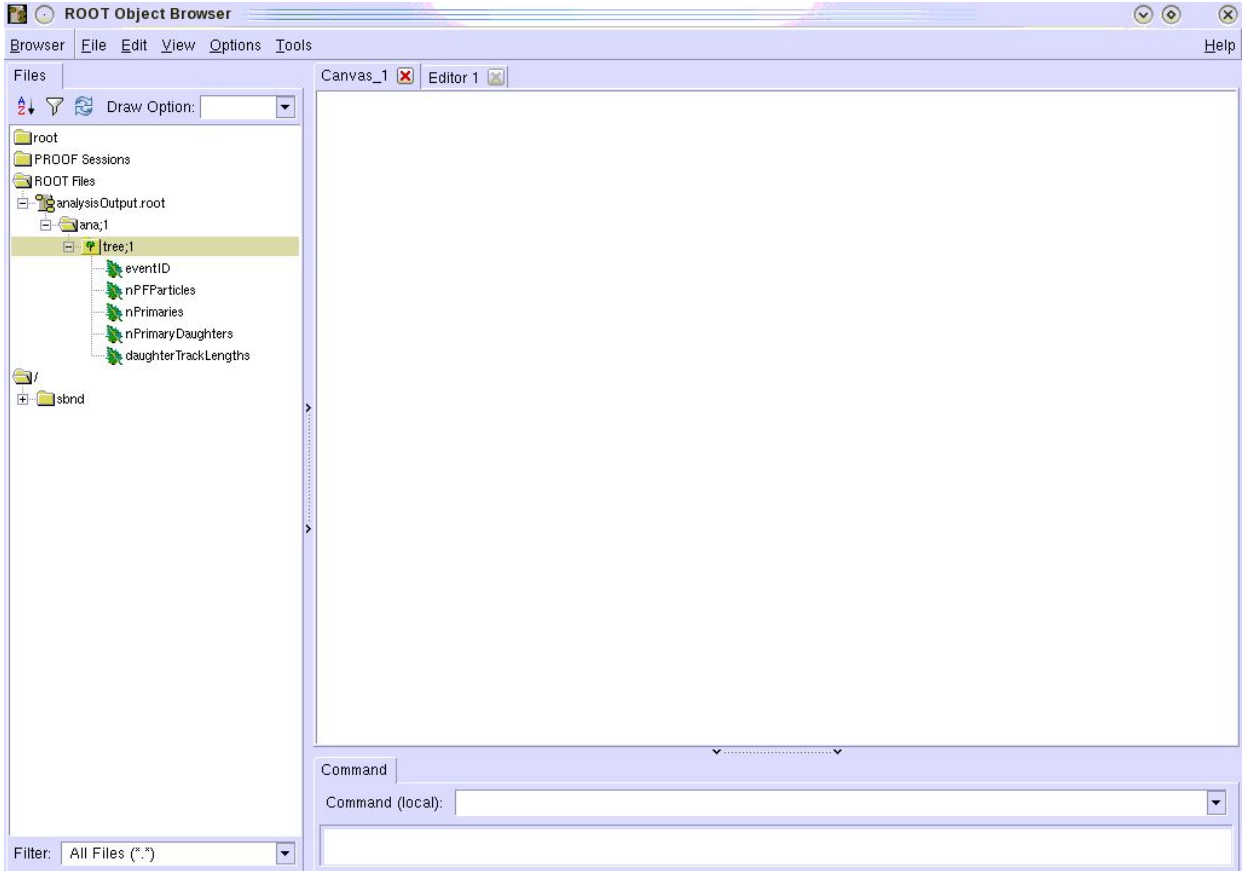
In your TBrowser

Hopefully you'll see something like this open up



In your TBrowser

Navigate into your file and find the tree

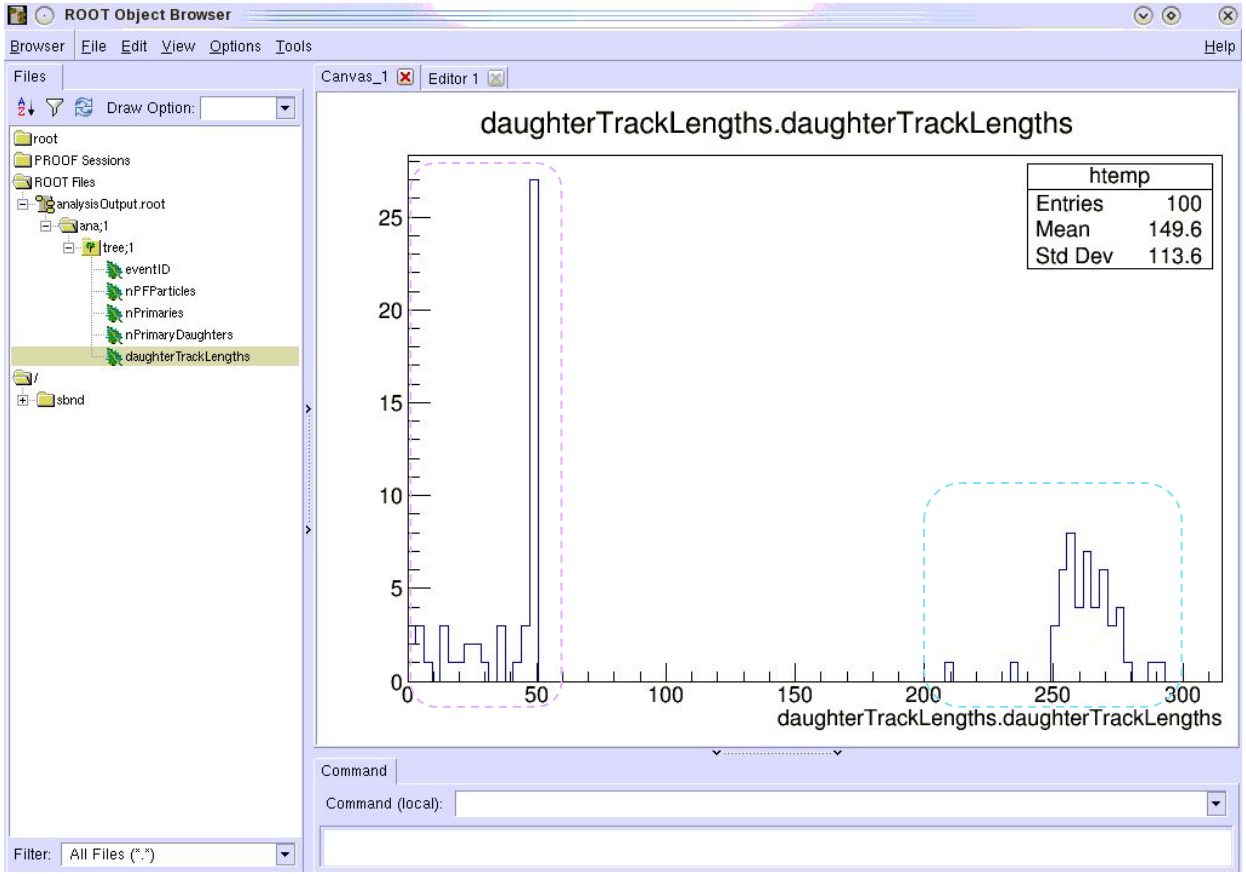


In your TBrowser

Open up the **`daughterTrackLengths`** branch

You can almost make out what is likely to be separate muon and proton distributions!

Probably with some amount of contamination



Let's write that histogram to our output file

Rather than creating a TTree then creating a histogram from the TTree we can create histograms in the analyser module

TTrees generally give greater flexibility but directly creating histograms can be useful in some cases

```
// ROOT includes
#include <TTree.h>
#include <TH1F.h>
```

```
// Create output TTree
TTree *fTree;
// Create output histogram
TH1F *fTrackLengthHist;
```

```
    // Add paramters from the track to the branch vector
    fDaughterTrackLengths.push_back(pfpTrack->Length());

    // Fill the histogram with the track lengths
    fTrackLengthHist->Fill(pfpTrack->Length());
```

```
// Get the TFileService to create the output TTree for us
art::ServiceHandle<art::TFileService> tfs;
fTree = tfs->make<TTree>("tree", "Output TTree");
fTrackLengthHist = tfs->make<TH1F>("trackLengthHist","Reconstructed track lengths; Track length [cm]",20,0,350);
```

Use what you've learnt so far to implement these lines in the appropriate places...

Check your work!

Compile and run!

Check that the output file now has a new entry:

```
root [2] .ls
TDirectoryFile*         ana     ana (AnalyzeEvents) folder
 KEY: TTree     tree;1  Output Tree
 KEY: TH1F      trackLengthHist;1       Reconstructed Track Lengths
```

Compare your histogram with the one you saw in the TTree.

They should be identical! (Up to maybe different binning)

The University Of Sheffield.

DEEP UNDERGROUND NEUTRINO EXPERIMENT

Associations: Going a little deeper

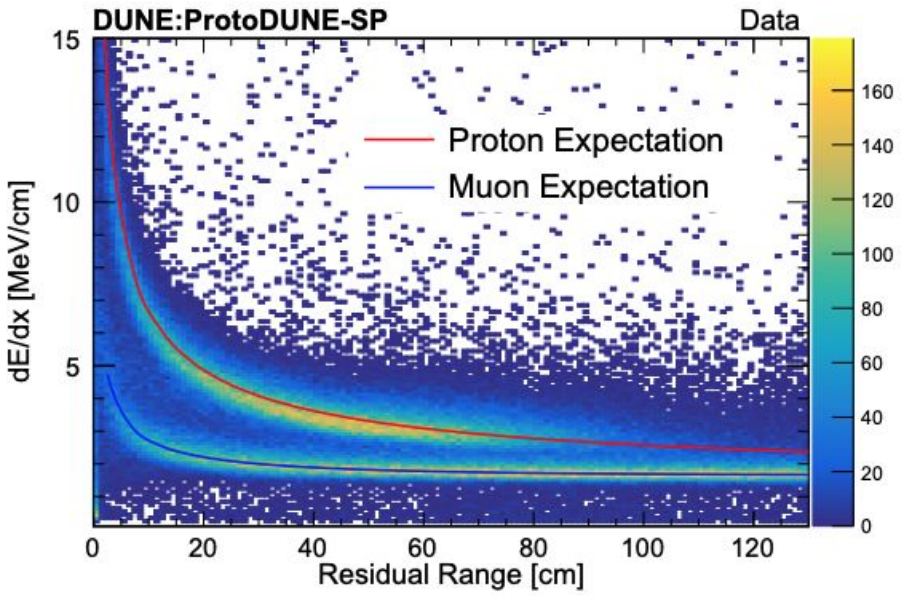
Particle Ionisation

A plot from ProtoDUNE-SP LArTPC showing the 2D dE/dx vs. residual range distributions for Muons and Protons produced in a test beam at CERN.

The theoretical distributions for each particle type are given by the lines.

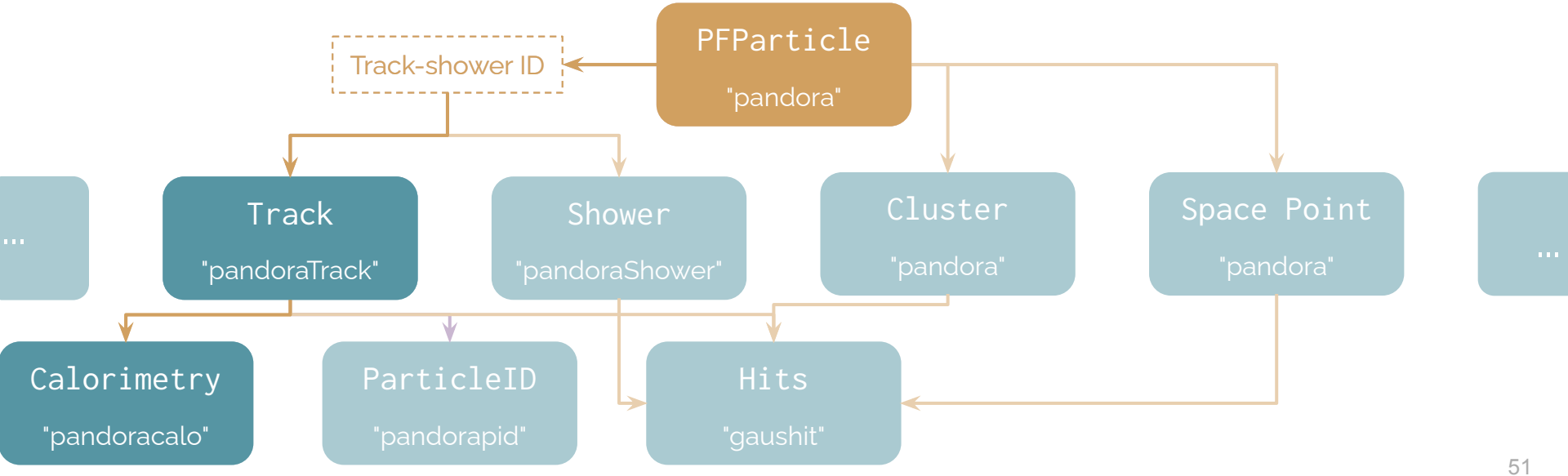
Good separation between Muons & Protons due the large difference in mass.

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



Accessing energy information

Associating the `anab::calorimetry` objects to `recob::Tracks` will give us energy information



What is a calorimetry object

We are now looking inside your loop over the `recob::Track` associations from `recob::PFParticles`

In contrast to the Tracks associated to the PFParticles where there was maximum 1 entry in the vector, the Calorimetry object can have 3: 1 for each plane in the detector.

As you did previously, define a vector of art pointers to the calorimetry objects & check if they're valid.

```
// Now access the calorimetry association for this track
const std::vector<art::Ptr<anab::Calorimetry>> trackCalos(trackCaloAssns.at(pfpTrack.key()));
```

Then you can loop over the calorimetry objects, make sure you can access the plane ID, and only look at the collection plane (plane number 2) for ease.

```
// Now loop over the calorimetry objects and select the one on the collection plane
for(const art::Ptr<anab::Calorimetry> &calo: trackCalos){

  // Get the plane number in a simple format
  const int planeNum(calo->PlaneID().Plane);

  // If it is not on the collection plane, skip it
  if(planeNum != 2)
    continue;

  // Add paramters from the calorimetry objects to the branch vector
  fDaughterTrackdEdx.push_back(calo->dEdx());
  fDaughterTrackResidualRange.push_back(calo->ResidualRange());
} // Calorimetry Track associations
```

The dE/dx & ResidualRange objects we want have entries for every trajectory point in the track and have type `std::vector<float>` See doxygen `for details`

This is great, we can pass the vector of dE/dx & ResidualRange objects directly to the vector (of vectors) we already defined!

How this is implemented

- We will use techniques you have already seen to access the calorimetry objects
  - With a couple of slight differences

- You once again need to
  - Add the relevant header for the `anab::Calorimetry` object
  - Add the module label to your configuration file and access it in the constructor
  - Add any declarations for new variables you want to push to your tree along with a new branch
  - Access the list of `anab::Calorimetry` objects from the list of `recob::Track` objects using `art::FindManyP`
  - If you are feeling confident have a go on your own now

Once again, the little bits before we analyze

In the configuration file

The **anab::Calorimetry** header

The module label and any other vectors of variables you want to declare and initialise
Notice these are **std::vectors** of **std::vectors**. It will become clear why this is the case shortly

Add the branches to the TTree
Despite these being **std::vectors** of **std::vectors** the syntax is exactly the same

In analysisConfig.fcl

```
CalorimetryLabel:    "pandoracalo"
```

In analyzeEvents_module.cc

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

```
std::vector<std::vector<float>> fDaughterTrackdEdx;
std::vector<std::vector<float>> fDaughterTrackResidualRange;

// Define input labels
const std::string fCalorimetryLabel;
```

```
fTree->Branch("daughterTrackdEdx",&fDaughterTrackdEdx);
fTree->Branch("daughterTrackResidualRange",&fDaughterTrackResidualRange);
```

I have purposefully left out some things you've seen before:

- Initialising `fCalorimetryLabel` in the constructor
- Clearing the vectors at the start of every event

See slide 37 for hints!

Inside the analyze function

- We now need to access the calorimetric associations to `recob::Tracks`, for this we need the `art_ptr_vector` of `recob::Tracks`
  - This is done using the same method as for the `recob::PFParticles`

```
// Load the Tracks from Pandora
art::Handle<std::vector<recob::Track>> trackHandle;
std::vector<art::Ptr<recob::Track>>    trackVec;
if(e.getByLabel(fTrackLabel, trackHandle))
  art::fill_ptr_vector(trackVec, trackHandle);
```

- We can then use `art::FindManyP` in the same way we did for `recob::PFParticles` and their associated `recob::Tracks`

```
art::FindManyP<anab::Calorimetry> trackCaloAssns(trackVec, e, fCalorimetryLabel);
```

Build, run, look at 2D histogram!

- I won't recall the way you build and run, hopefully that's clear from previous slides/times you've done it
- But I will show you how to quickly plot a 2D histogram in ROOT

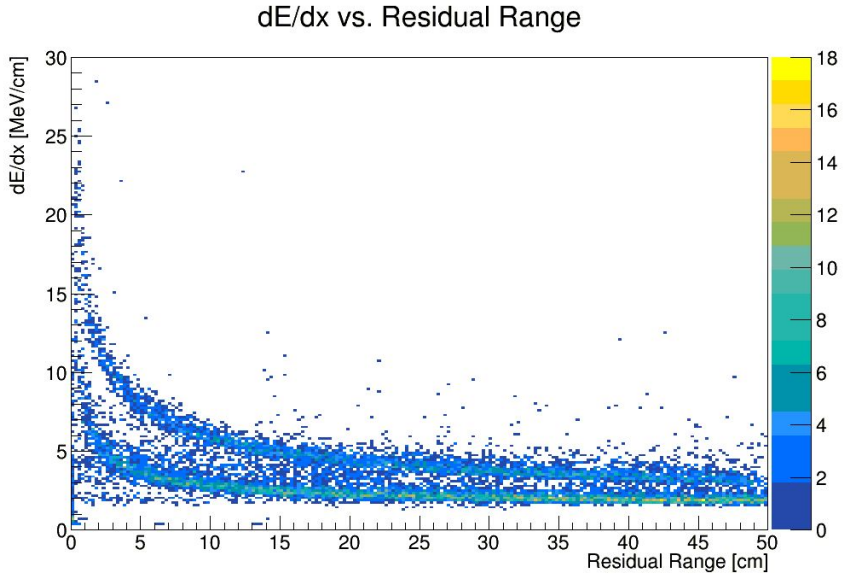
```
root[0] ana->cd()
```

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

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

You should see something like this!

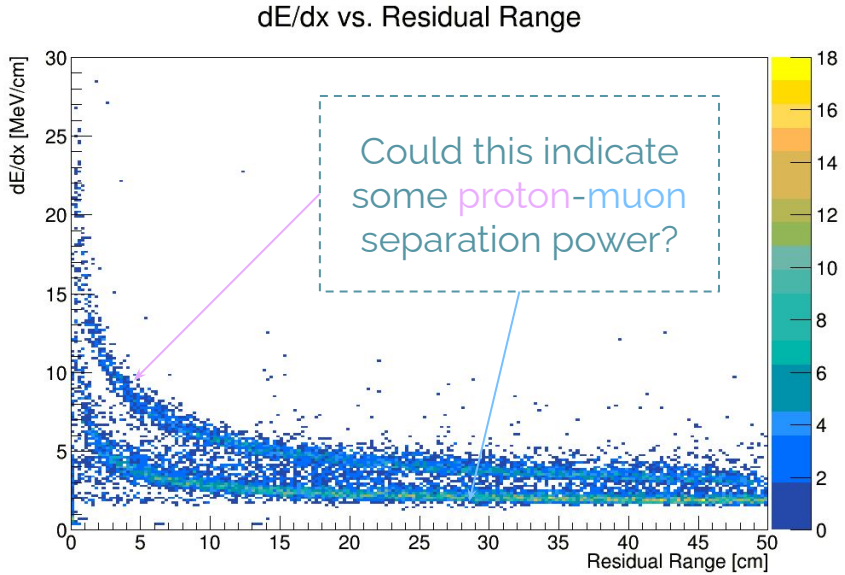
What do you find most interesting about the distribution?



Please note that I added the axes labels myself in the canvas window

You should see something like this!

What do you find most interesting about the distribution?



We'll try and get to the bottom of this now

A **very** simple PID

Dom, start a timer

Finding the longest track

- Since we have generated a single muon and proton with defined momenta, we can be reasonably confident that they will be very different lengths in each event

- So! Let's use this as a very simple particle identification technique for our sample

- We need to loop over all the `recob::Tracks` associated to the `recob::PFParticles` which are daughters of the neutrino once again, but we'll do this independently from our main analysis loop

Finding the longest track

Declare a vector of booleans
Add a corresponding branch to your tree

```
std::vector<bool> fDaughterLongestTrack;
```

Start by initialising a float to be unphysically small as the **`longest`** length and an invalid ID integer as the initial ID of the longest track, **`longestID`**

```
// Search for the longest daughter ID
int longestID       = -1;
float longestLength = std::numeric_limits<float>::lowest();
for(const art::Ptr<recob::PFParticle> &pfp: pfpVec){
  // We are only interested in the neutrino daughter particles
  if(pfp->Parent() != neutrinoID)
    continue;

  // Get the tracks associated with this PFParticle
  const std::vector<art::Ptr<recob::Track>> pfpTracks(pfpTrackAssns.at(pfp.key()));

  // There should only ever be 0 or 1 tracks associated with a single PFParticle
  if(pfpTracks.size() == 1){

    // Get the first element of the vector
    const art::Ptr<recob::Track> &pfpTrack(pfpTracks.front());
```

In a standalone loop over the neutrino daughter tracks
If the current track length is longer than the **`'longest'`**: Redefine **`longest`** to be that track length and the **`longestID`** to be the ID of that track

```
    // If this track is the longest, save the ID and set the length of the longest track
    if(pfpTrack->Length() > longestLength){
      longestID     = pfpTrack->ID();
      longestLength = pfpTrack->Length();
    } // Length check
  } // PFParticle Track
} // PFParticles
```

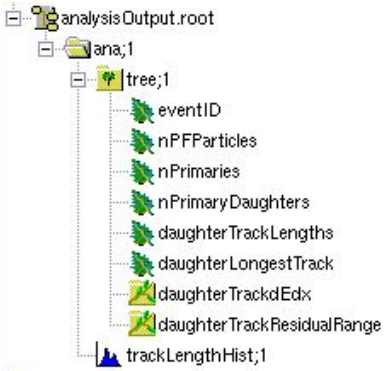
Fill the boolean vector in your main analysis loop

```
// Add paramters from the track to the branch vector
fDaughterTrackLengths.push_back(pfpTrack->Length());
fDaughterLongestTrack.push_back(pfpTrack->ID() == longestID);
```

Current status of your output tree

Your new list of branches should look something like this:
Once again this is looking inside the TBrowser

The added vector of booleans means we can now look at each track-based variable with `conditional formatting:`
Check if each track is the longest in the event in the Draw function




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

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

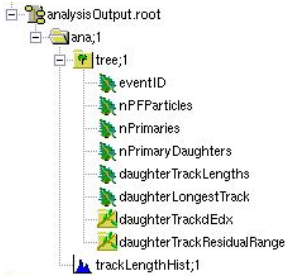
```
root[2] TH2D *hShort = new TH2D("hShort","dE/dx vs. Residual Range", 200, 0, 50, 200, 0, 30)
```

Current status of your output tree

Drawing the 2 histograms with the relevant conditions:

```
root[3] tree->Draw("daughterTrackdEdx:daughterTrackResidualRange>>hLong", "daughterLongestTrack", "")
```

```
root[4] tree->Draw("daughterTrackdEdx:daughterTrackResidualRange>>hShort", "!daughterLongestTrack", "same")
```



Changing the marker colours so we can distinguish between the 2!

```
root[5] hLong->SetMarkerColor(kViolet)
```
Alternative colour options are here: https://root.cern.ch/doc/master/classTColor.html

```
root[6] hShort->SetMarkerColor(kBlue)
```

```
root[6] c1->Modified()
```
Tell the canvas (default c1) to implement these changes and redraw the canvas

We'll see how this affects both your energy and track length plots next!

The University Of Sheffield.

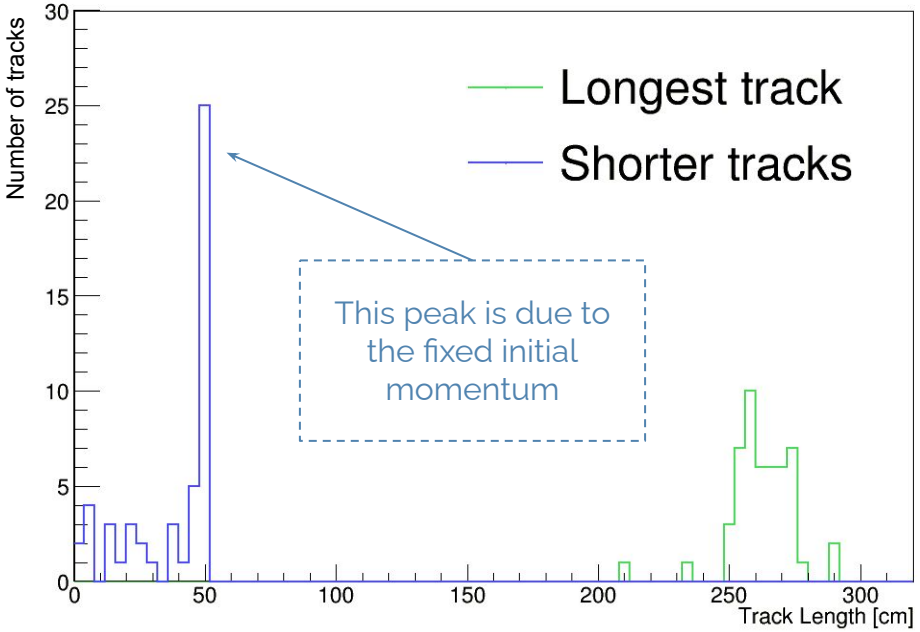
DEEP UNDERGROUND NEUTRINO EXPERIMENT

Let's look at some final plots

Track lengths

A quick comparison of track lengths for the longest track and everything else confirms there is never any ambiguity within a single event as to which track might be the muon.

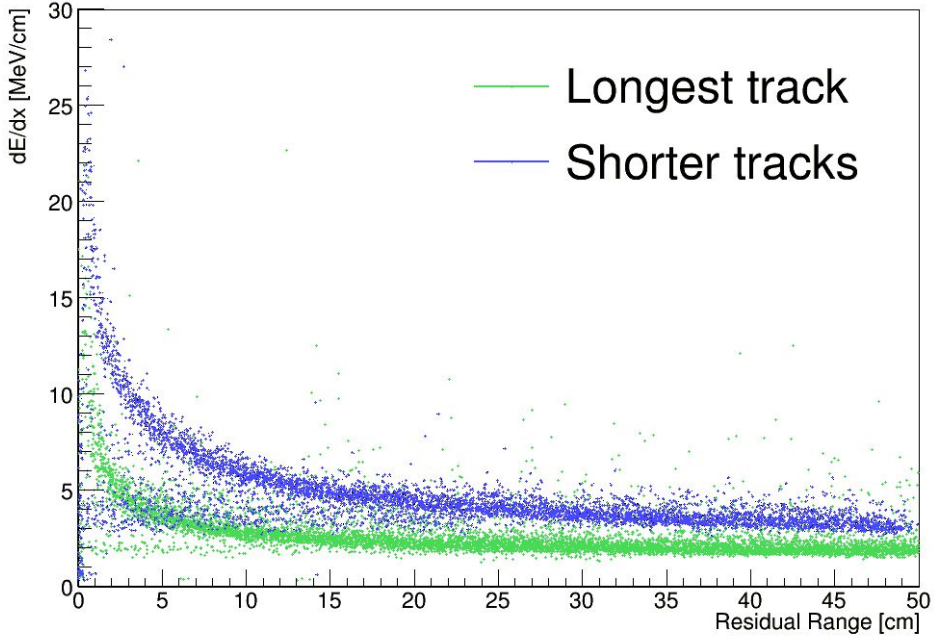
The longest track is always significantly longer than everything else.



Energy distributions

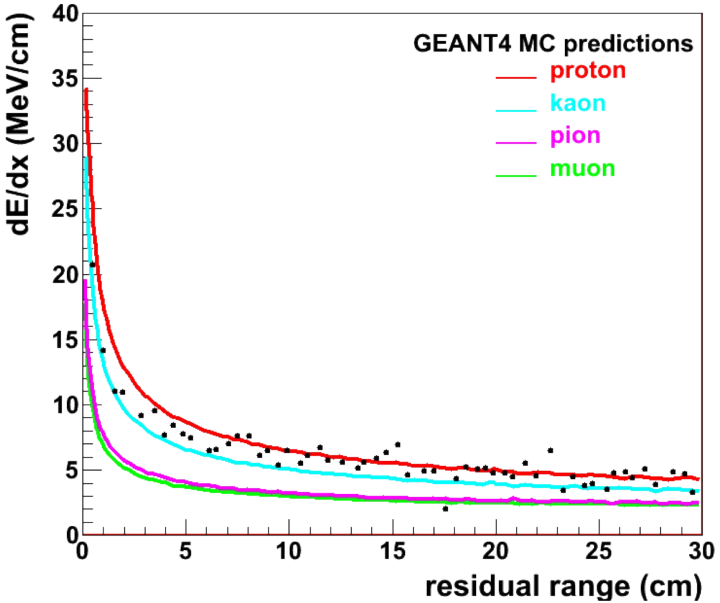
An even better indication of particle flavour occurs when we plot the dEdx vs residual range of the tracks.

Here you can see there is a reasonably clear separation between the longest and shorter tracks!

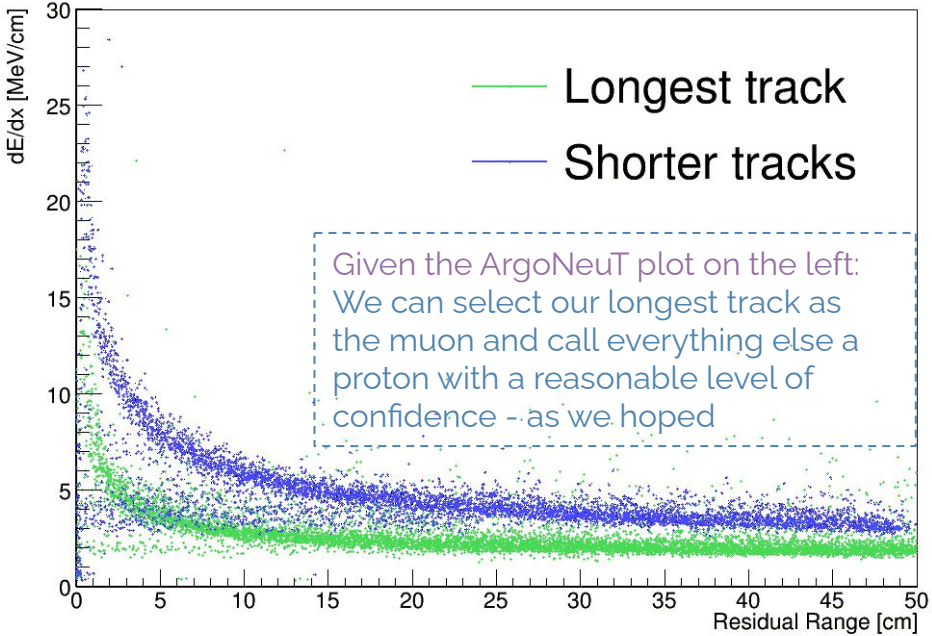


Energy distributions

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



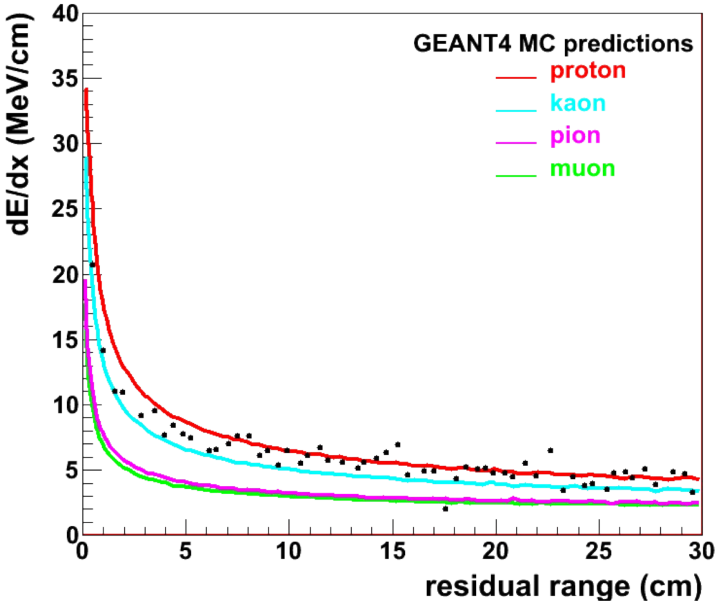
Plot from ArgoNeuT showing the theoretical separating power of the average dE/dx vs. residual range distributions.
The theoretical distributions for each particle type are given in varying colours, the energy loss of a stopping track in the ArgoNeuT detector is shown by the black dots



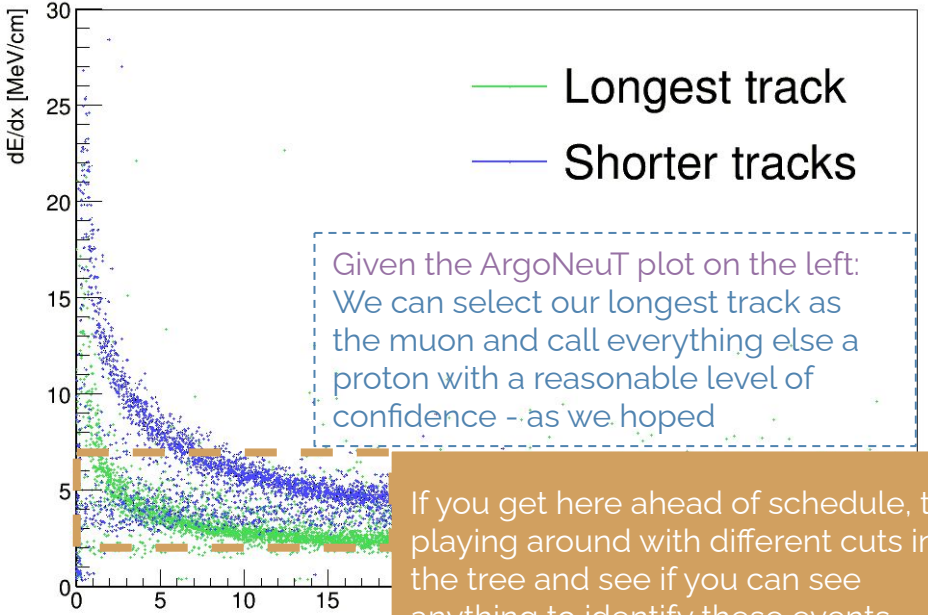
Given the ArgoNeuT plot on the left:
We can select our longest track as the muon and call everything else a proton with a reasonable level of confidence - as we hoped

Energy distributions

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



Plot from ArgoNeuT showing the theoretical separating power of the average dE/dx vs. residual range distributions.
The theoretical distributions for each particle type are given in varying colours, the energy loss of a stopping track in the ArgoNeuT detector is shown by the black dots



Given the ArgoNeuT plot on the left:
We can select our longest track as the muon and call everything else a proton with a reasonable level of confidence - as we hoped

If you get here ahead of schedule, try playing around with different cuts in the tree and see if you can see anything to identify these events


The
University
Of
Sheffield.


DEEP UNDERGROUND
NEUTRINO EXPERIMENT

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)

Documentation and additional information

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

- `recob::PFParticle` - https://nusoft.fnal.gov/larsoft/doxsvn/html/classrecob_1_1PFParticle.html
- `art::FindManyP` - https://nusoft.fnal.gov/larsoft/doxsvn/html/classart_1_1FindManyP.html
- `recob::Track` - https://nusoft.fnal.gov/larsoft/doxsvn/html/classrecob_1_1Track.html
- `anab::Calorimetry` - https://nusoft.fnal.gov/larsoft/doxsvn/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 important file locations

My version of the code lives here:

```
$MRB_SOURCE/duneana/duneana/Workshop/Analysis/.FinishedModule/AnalyzeEvents_module.cc
```

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

```
$MRB_SOURCE/duneana/duneana/Workshop/Analysis/.FinishedModule/run_analyzeEvents.fcl
```

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

Please note:

There is some additional material on the following slides for anyone who finishes early. These also contain some versions of the code with additional functionality and refactoring the code to make it more modular and efficient.

Previous tutorials (SBND-based)

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/getFile.py/access?contribId=12&sessionId=4&resId=0&materialId=slides&confId=5856

Rhiannon Jones' tutorial from 2019 is here:
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:
https://indico.hep.manchester.ac.uk/getFile.py/access?contribId=13&sessionId=2&resId=0&materialId=slides&confId=5372

Final final remarks

There is a refactored version of the module that modularises the code and code and makes it more efficient and readable

```
$MRB_SOURCE/duneana/duneana/Workshop/Analysis/.FinishedModule/.ReorderedModule
```