Módulo de Análise

I Escola Brasileira de Neutrinos

Curitiba, 7 de Agosto de 2025

Marcelo Oliveira
ismerio@pos.if.ufrj.br
@Marcelo Oliveira no DUNE Slack

Traduzido e adaptado do material produzido por Alex Wilkinson e Isobel Mawby para o CERN LArSoft Wokshop em 2025

Disclaimer

Estes slides, bem como o conteúdo apresentado, foram **muito marginalmente** adaptados e traduzidos de versões anteriores de Workshops e Escolas produzidas pelo grupo DUNE UK. Em particular, iremos usar a versão de código da última versão CERN Workshop 2025. Nosso muito obrigado a todos que cuidadosamente montaram este curso!

Disclaimer 2

- Possivelmente, o tempo não será suficiente para cobrir tudo neste módulo. Vamos tentar, mas não tenha pressa!
- Esperamos que a apresentação esteja estruturada de maneira que você consiga acompanhar os exercícios em seu próprio tempo.
- Então não se preocupe! É muita coisa mesmo!
- Se você está acessando os slides via pdf, talvez queira acessar via Google Slides diretamente, alguns blocos de código renderizam melhor lá.

Objetivos

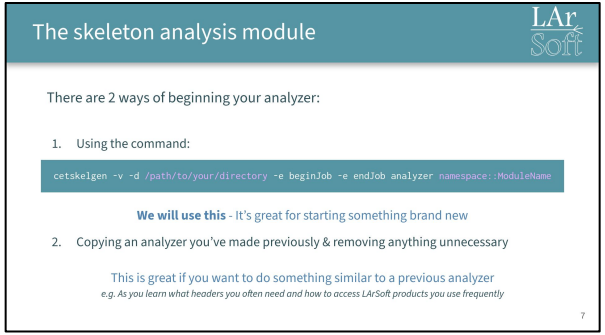
- Aprender a extrair um pouco da física dos eventos que reconstruímos.
  - Não se preocupe se você não tem um arquivo de reconstrução, temos um para você!
- Aprender a acessar a informação reconstruída.
  - Existe um procedimento genérico para acessar praticamente qualquer informação nos arquivos que você produziu.
- Vamos focar em:
  - Objetos reconstruídos produzidos pelo Pandora.
  - Associações entre esses objetos e informações de alto nível.
- Não tenha pressa, tente entender bem tudo o que for fazendo.
- Vamos produzir alguns plots!

Estrutura da apresentação

Novo Tópico

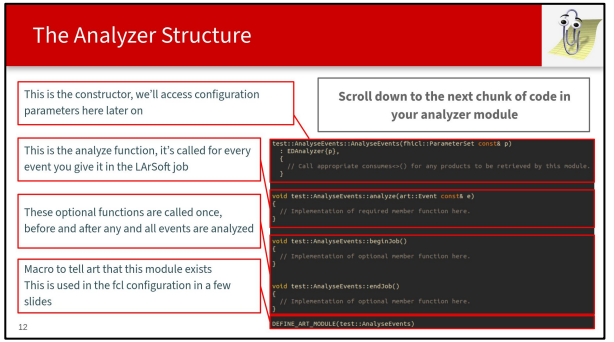
Vamos fazer uma pausa a cada tópico para garantir que todos estejam acompanhando!

Explicação



Detalhes de sintaxe, introdução de novos comandos, dicas e informações.

Exercício



Mão na massa! Vamos aplicar o que formos aprendendo.

Obs: slides com  possuem suporte para seleção (o código pode ser copiado!).

1. Como começar um módulo de análise

Como começar um módulo de análise

Existem duas formas de começar um módulo de análise:

1. “Do zero”, criando um esqueleto com:

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

Vamos utilizar esta forma.

2. Copiando um módulo já existente, fazendo alterações e retirando o que é desnecessário.

Ótima opção se você deseja fazer algo muito parecido com um código já existente.

ex.: Conforme você aprender quais headers você precisa e como acessar os produtos do LArSoft.

cetskelgen

Estas são funções opcionais que serão adicionadas. Vamos vê-las em detalhes em breve.

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

Para mais informações, visite:
`https://cdcvs.fnal.gov/redmine/projects/cetlib/wiki/Cetskelgen`

Escolha algo que faça sentido aqui, ex.: **`test::AnalyseEvents`**

Sua vez!

Vamos setar o ambiente se já não tiver feito isso:

```
source /cvmfs/dune.opensciencegrid.org/products/dune/setup_dune.sh
source localProducts*/setup
mrbslp
```

1. Vá para:

Dê uma olhada nos arquivos `CMakeLists.txt` and `build.sh`

```
cd $MRB_SOURCE/duneana/duneana/CERNWorkshop/Analysis
```

2. Utilize o comando cetskelgen:

Aqui é onde se coloca o diretório no qual o arquivo será produzido, no caso, nesta pasta em que estamos

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

A estrutura do código

- Você agora encontrará um arquivo `AnalyseEvents_module.cc`, este é o seu *analyser*!
- Abra-o.
- Ele deverá ter esta estrutura:

```
////////////////////////////////////////////////////////////////////////
// Class:       AnalyseEvents
// Plugin Type: analyzer (Unknown Unknown)
// File:        AnalyseEvents_module.cc
//
// Generated at Tue Jul 22 12:17:06 2025 by Marcelo Oliveira 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;
```

A estrutura do código

Aqui temos o construtor padrão da classe. Vamos acessar parâmetros de configuração aqui logo mais.

Role para baixo para ver o próximo bloco de código

Esta é a função de análise. Ela será chamada para cada evento do seu job.

Estas funções são opcionais e são chamadas antes e depois de todos os eventos serem analisados.

Aqui temos uma Macro para declarar para o art que a função existe. Será usada no fcl a seguir.

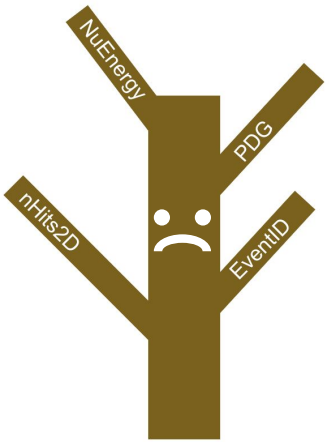
```
test::AnalyseEvents::AnalyseEvents(fhicl::ParameterSet const& p)
 : EDAnalyzer{p}  // ,
 // More initializers here.
{
 // Call appropriate consumes<>() for any products to be retrieved by this
module.
}

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)
```

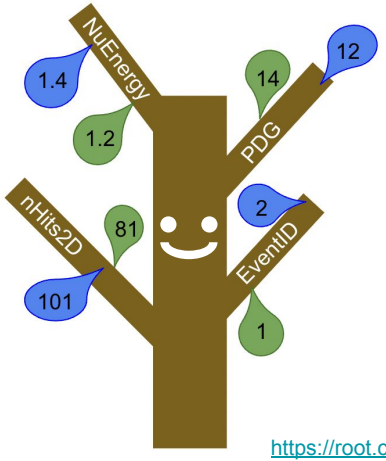
2. Obtendo informações da análise

Salvando informações para análise

1) Vamos criar uma TTree para armazenar informações dos nossos eventos.

2) Preencher a TTree funciona como um loop.



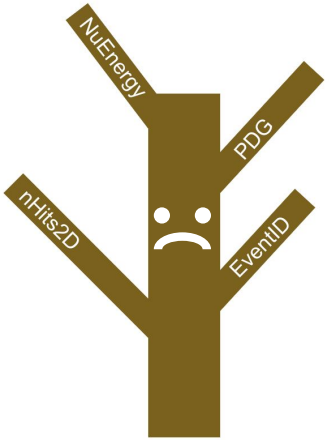


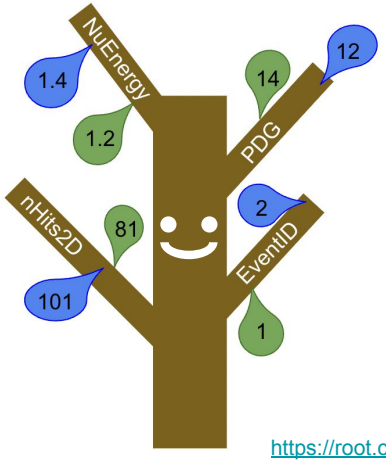
https://root.cern.ch/doc/master/classTTree.html

Salvando informações para análise

3) Vamos preenchê-la para **todos os eventos** e, portanto, cada entrada corresponderá a um evento.

4) Lembre-se que a função analyze é chamada para cada evento, então só precisamos preencher a TTree ao final da função.





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

Criando a TTree

Vamos adicionar os headers do LArSoft e do ROOT

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

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

Precisamos declarar a TTree

```
private:

  // Create output TTree
  TTree *fTree;
};
```

E criá-la

```
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");
}
```


Nota: a ordem representa a localização no arquivo.

Preenchendo a TTree

Vamos declarar as variáveis que iremos olhar evento a evento.

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

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

Aqui, acessamos a variável do evento e preenchemos a TTree.

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

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

Vamos criar um Branch para esta variável que estamos salvando.

```
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);
}
```

Nota: a ordem representa a localização no arquivo

Como rodar o módulo

Para de fato rodar o módulo de análise, precisamos criar dois arquivos fcl:

- O primeiro será de configuração - **um fcl de inclusão (*include*)**
  - Aqui vamos apontar para o módulo de análise os objetos e parâmetros que queremos acessar nos arquivos de input (fará mais sentido nos próximos slides)

- O segundo será utilizado para rodar análise **- um fcl de execução (*run/job*)**
  - É isto que iremos chamar com o *lar* e é o que unirá o arquivo de configuração com o módulo que acabamos de construir.

- O principal motivo pelo qual não definimos nossos parâmetros diretamente no fcl de execução é que vários fcls de execução podem herdar todos de um mesmo fcl de inclusão.

fcl 1: Configurando o analyzer

fcl 1: Configuração. Crie um arquivo chamado **`analysisConfig.fcl`** e preencha-o com:

Um nome para este conjunto de parâmetros.

Mais detalhes do que isto faz aqui.

```
BEGIN_PROLOG

analyseEvents:
{
  module_type:  "AnalyseEvents"
}

END_PROLOG
```

Aqui precisa ser o nome que demos ao módulo de análise.

É aqui que vamos adicionar mais configuráveis em nosso módulo mais tarde.

fcl 2: Executando o código

Crie outro arquivo, agora chamado run_analyseEvents.fcl

Inclua o seu fcl de configuração

Dê um nome a este processo

Não pode ter "_"

Vamos configurar que o input será um arquivo root

Aqui será o nome do output

Lembre-se que você poderá mudá-lo com -o na linha de comando

ana coloca nosso módulo **analyseEvents** como parte do workflow

Note que colocamos o mesmo nome que escolhemos no arquivo de configuração

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

process_name: AnalyseEvents

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

services:
{
  TFileService: { fileName: "analysisOutput.root" }

  @table::dunefd_1x2x6_simulation_services # 1x2x6 HD detector
}

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

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

Vamos tentar rodar…

Arquivo de reconstrução

Não tem um arquivo de reconstrução? Sem problemas!

Você pode utilizar este que preparamos. Basta executar:

```
wget --no-check-certificate
https://indico.ggte.unicamp.br/event/13/sessions/16/attachments/168/294/prod_1mu1p_100evts_g4_detsim_reco1_reco2.root
```

Compilando e rodando

Primeiro, precisamos compilar o que escrevemos até agora:

Do diretório `$MRB_SOURCE/duneana/duneana/Workshop/Analysis`:

```
source build.sh
```

Dê uma olhada no que este script está fazendo e veja se está confortável com que cada um dos comandos faz

Quando terminar, execute o seu módulo!

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

Aqui vamos testar com somente 10 eventos para ver se está tudo certinho, depois vamos aumentar a estatística

Output esperado

```
22-Jul-2025 13:17:58 CDT  Opened input file "input_file.root"
Begin processing the 1st record. run: 20000011 subRun: 0 event: 1 at 22-Jul-2025 13:17:58 CDT
Begin processing the 2nd record. run: 20000011 subRun: 0 event: 2 at 22-Jul-2025 13:17:59 CDT
Begin processing the 3rd record. run: 20000011 subRun: 0 event: 3 at 22-Jul-2025 13:17:59 CDT
Begin processing the 4th record. run: 20000011 subRun: 0 event: 4 at 22-Jul-2025 13:17:59 CDT
Begin processing the 5th record. run: 20000011 subRun: 0 event: 5 at 22-Jul-2025 13:17:59 CDT
Begin processing the 6th record. run: 20000011 subRun: 0 event: 6 at 22-Jul-2025 13:17:59 CDT
Begin processing the 7th record. run: 20000011 subRun: 0 event: 7 at 22-Jul-2025 13:18:00 CDT
Begin processing the 8th record. run: 20000011 subRun: 0 event: 8 at 22-Jul-2025 13:18:00 CDT
Begin processing the 9th record. run: 20000011 subRun: 0 event: 9 at 22-Jul-2025 13:18:00 CDT
Begin processing the 10th record. run: 20000011 subRun: 0 event: 10 at 22-Jul-2025 13:18:00 CDT
22-Jul-2025 13:18:00 CDT  Closed input file "input_file.root"
%MSG-i NuRandomService:  AnalyseEvents:ana@EndJob 22-Jul-2025 13:18:00 CDT  ModuleEndJob

Summary of seeds computed by the NuRandomService
Random policy: 'perEvent'
  algorithm version: EventTimestamp_v1

%MSG

TrigReport ---------- Event summary -------------
TrigReport Events total = 10 passed = 10 failed = 0

TrigReport ---------- Modules in End-path ----------
TrigReport       Run     Success         Error Name
TrigReport        10       10               0 ana

TimeReport ---------- Time summary [sec] -------
TimeReport CPU = 0.273616 Real = 1.808679

MemReport  ---------- Memory summary [base-10 MB] ------
MemReport  VmPeak = 2061.36 VmHWM = 634.708

Art has completed and will exit with status 0.
```

Eventos estão saudáveis :)

Algumas informações de performance. Cuidado quando for rodar algo com muitos eventos. As gpvms são de uso comum!

Compilando e rodando

Primeiro, precisamos compilar o que escrevemos até agora:

Do diretório `$MRB_SOURCE/duneana/duneana/Workshop/Analysis`:

```
source build.sh
```

Dê uma olhada no que este script está fazendo e veja se está confortável com que cada um dos comandos faz

Quando terminar, execute o seu módulo!

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

Aqui vamos testar com somente 10 eventos para ver se está tudo certinho, depois vamos aumentar a estatística

Veja que o arquivo root de output foi criado. Vamos ver o que tem nele:

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

Abrindo o arquivo de output

Aqui podemos ver que o nome do nosso analyzer no fcl de execução é o nome do diretório (**ana**): Abra-o com **->cd()**

Aqui vemos a TTree que criamos, podemos usar **Scan()** para ver seus conteúdos (também pode ser **Show(*entryNumber*)**, **TBrowser**, etc...)

Sua TTree existe e contém os eventIDs! Sucesso!

```
$ root -l analysisOutput.root
root [0]
Attaching file analysisOutput.root as _file0...
(TFile *) 0x44c3710
root [1] .ls
TFile**         analysisOutput.root
 TFile*         analysisOutput.root
  KEY: TDirectoryFile   ana;1   ana (AnalyseEvents) folder
root [2] ana->cd()
(bool) true
root [3] .ls
TDirectoryFile*           ana    ana (AnalyseEvents) 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 *
************************
```

Tente fazer também um : root [5] new TBrowser

3. Acessando outros data products

Vamos acessar outras variáveis e melhorar o nosso módulo de reconstrução!

Acessando outras informações

- Até agora só temos os eventIDs, como faríamos para acessar informações de outros estágios, como por exemplo do Pandora?
- A informação pode ser guardada de duas formas:

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

        {PFP_A,    PFP_B,     PFP_C}
```

1) Vetor de objetos:

ex.: um vetor de todas as `PFParticles` criadas pelo Pandora

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

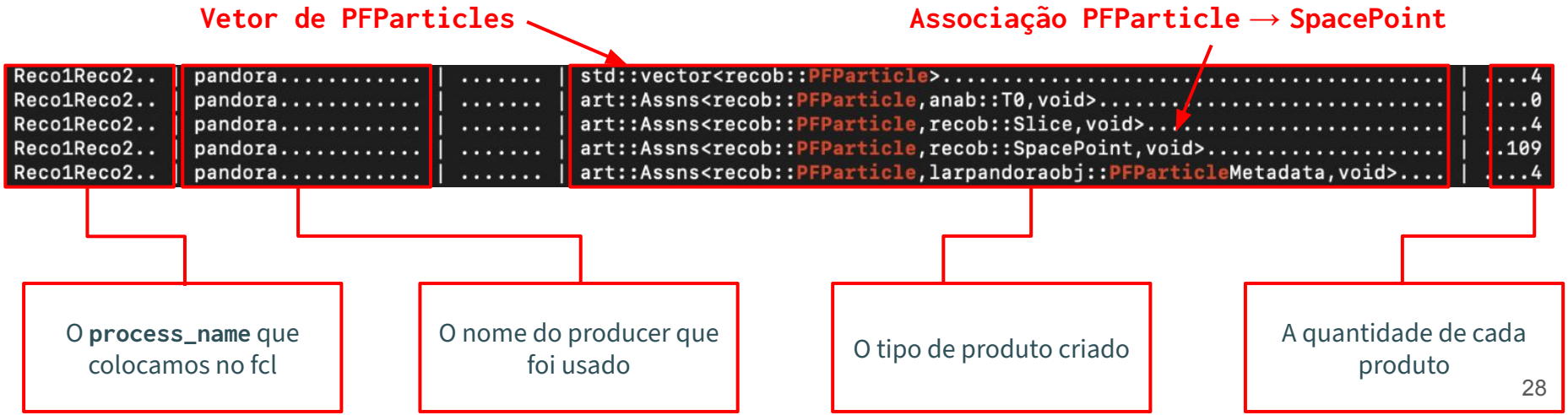
2) Associações:

ex.: links entre `PFParticles` e seus vértices reconstruídos

Acessando outras informações

- Podemos usar eventdump.fcl para ver quais data products estão salvos em nossos arquivos.

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



Acessando Vetores

- Slice: é um conjunto de hits que é agrupado por parecerem vir de uma mesma interação física.
- Vamos obter o vetor de slices.
- Precisamos setar o **data object handle** - este é o link entre o seu código e o vetor nos arquivos de simulação/dados.

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

O tipo de objeto que queremos

e é o art::Event em que estamos

O nome do producer que criou o data object

- Depois que garantimos que o handle é válido, podemos recuperar o vetor em nosso código:

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

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

Associações - Detalhes Técnicos

- Vamos obter o vetor de `PFParticles` de um dado slice

- Primeiro iniciamos o objeto `FindManyP`, que é o link entre o seu código e as associações de um dado vetor e objetos (neste caso, o vetor em que no slice vive)

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

nosso handle para o vetor de objetos

O nome do producer que criou a associação

Associações - Detalhes Técnicos

- Assim, para obter as `PFParticles` associadas a um slice, neste caso, o primeiro no vetor `sliceVector`:

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

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

O que seria esta função key()?

O que é essa função key?

- Todo `art::Ptr<...>` tem uma função key
- Ela retorna o índice do objeto que está sendo apontado no vetor em que ele vive, e é utilizada para identificar as associações.

Considere:

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

Então:

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

Assim, para obter o vetor de `PFParticles` associado ao `sliceC`, faríamos:

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

4. Investigando a hierarquia

Obtendo a hierarquia do neutrino

- Em princípio, nosso experimento pode ter background de raios cósmicos, então o output da reconstrução será constituído de slices, alguns contendo hierarquias tipo rc ou tipo neutrino.
- Uma das melhores formas de se obter a hierarquia de `PFParticles` para neutrinos é esta:

```
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;

        // We have found our neutrino!
      }
  }
```

O Pandora assume que o PDG de um PFP neutrino seja 12 ou 14, mas NUNCA use esse número para separação de sabores

Implementando variáveis para a hierarquia

- Vamos “calcular” algumas variáveis relacionadas à hierarquia e adcioná-las à TTree.

1) Primeiro, vamos precisar destes includes:

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

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

2) E criar novos membros e seus respectivos Branches na TTree:

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

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

Lendo as variáveis de hierarquia

Vamos escrever para praticar!

A cada evento, inicializamos nossas variáveis com zeros.

Obtemos aqui os slices do evento e as associações de `PFParticles`

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

  // Prepare variables for new event (reset counters to 0 / set default values / empty vectors)
  fNPFParticles = 0; fNPrimaryChildren = 0;

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

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

  // Get associations between slices and pfparticles & opt0 results
  art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);
```

Lendo as variáveis de hierarquia

Continuando logo abaixo do bloco anterior:

Vamos criar uma variável temporária para o nosso loop principal

Fazemos um loop pelos slices até que encontremos um `PFParticle` que seja um neutrino (aqui assumimos que só haja um candidato - não é necessariamente sempre verdade!)

Preenchemos as variáveis que criamos, e anotamos o ID do neutrino (e do slice)

```
// Filling our neutrino hierarchy variables
int nuID = -1;

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;

        // We have found our neutrino!

        nuID = slicePFP->Self();
        fNPFParticles = slicePFPs.size();
        fNPrimaryChildren = slicePFP->NumDaughters();

        break;
      }

    if (nuID >= 0)
      break;
  }

if(nuID < 0)
  return;

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

Novamente: aqui estamos supondo apenas um candidato. Em uma análise mais sofisticada, consideramos todos!

Condição se não houver candidato

Ler as variáveis de hierarquia

O bloco completo da função analyze ficar assim:

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

  // Prepare variables for new event (reset counters to 0 / set default values / empty vectors)
  fNPFParticles = 0; fNPrimaryChildren = 0;

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

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

  // Get associations between slices and pfparticles & opt0 results
  art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);

  // Filling our neutrino hierarchy variables
  int nuID = -1;

  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;

          // We have found our neutrino!

          nuID = slicePFP->Self();
          fNPFParticles = slicePFPs.size();
          fNPrimaryChildren = slicePFP->NumDaughters();

          break;
        }

      if (nuID >= 0)
        break;
    }

  if(nuID < 0)
    return;

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

Ler as variáveis de hierarquia

O bloco completo da função analyze ficar assim:

Mas atenção: ainda não definimos o que é `fSliceLabel`.

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

  // Prepare variables for new event (reset counters to 0 / set default values / empty vectors)
  fNPFParticles = 0; fNPrimaryChildren = 0;

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

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

  // Get associations between slices and pfparticles & opt0 results
  art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);

  // Filling our neutrino hierarchy variables
  int nuID = -1;

  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;

          // We have found our neutrino!

          nuID = slicePFP->Self();
          fNPFParticles = slicePFPs.size();
          fNPrimaryChildren = slicePFP->NumDaughters();

          break;
        }

      if (nuID >= 0)
        break;
    }

  if(nuID < 0)
    return;

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

Ler as variáveis de hierarquia

O bloco completo da função analyze ficar assim:

Mas atenção: ainda não definimos o que é `fSliceLabel`.

Precisamos disso para indicar para a associação qual módulo gerou o objeto que desejamos, no caso, o Pandora.

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

  // Prepare variables for new event (reset counters to 0 / set default values / empty vectors)
  fNPFParticles = 0; fNPrimaryChildren = 0;

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

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

  // Get associations between slices and pfparticles & opt0 results
  art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);

  // Filling our neutrino hierarchy variables
  int nuID = -1;

  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;

          // We have found our neutrino!

          nuID = slicePFP->Self();
          fNPFParticles = slicePFPs.size();
          fNPrimaryChildren = slicePFP->NumDaughters();

          break;
        }

      if (nuID >= 0)
        break;
    }

  if(nuID < 0)
    return;

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

Nunca utilize nomes de módulos fixos (hard-coded); isso dificulta a manutenção e a reutilização do código!

```
// Get associations between slices and pfparticles & opt0 results
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, "pandora");
```

```
// Get associations between slices and pfparticles & opt0 results
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);
```

Salve-os como variáveis membras!

Vamos ver como fazemos isto nos próximos slides…

Implementando as variáveis de hierarquia

- Passamos os nomes dos módulos de análise para o analyzer vía fcl de configuração `analysisConfig.fcl`:

Em analyzer:

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

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

Em `analysisConfig.fcl`:

```
BEGIN_PROLOG

analyseEvents:
{
  module_type:  "AnalyseEvents"

  SliceLabel: "pandora"
}

END_PROLOG
```

Hora da verdade

```
source build.sh
```

Compilar as mudanças

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

Rodar o analyzer

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

Checar o output

43

O que mudou

- Nossa TTree agora deve ter 2 novas Branches

nPFParticles mostra quantas partículas foram reconstruídas

nPrimaryChildren é o número de partículas primárias (filhas do neutrino) que foram reconstruídas

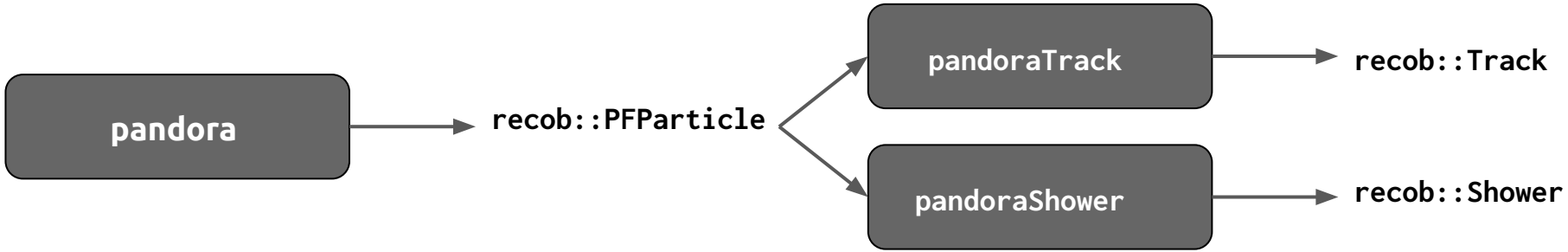
- Parece bastante razoável :)

```
[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. Adicionando informação das trajetórias (tracks)

Vamos dar uma olhada nas tracks de múons/prótons

No nosso workflow, todas as PFParticles são ajustadas como tracks e showers.



- A associação que queremos é:

`recob::PFParticle → recob::Track`

- Mas antes, precisaremos do handle da `PFParticle` para inicializar o `FindManyP`

Em detalhes

No bloco anterior da função analyze, declare nuSliceKey e atualize seu valor de acordo.

Alguns compiladores reclamam se não utilizamos as variáveis então estamos fazendo isso agora.

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

  // Prepare variables for new event (reset counters to 0 / set default values / empty vectors)
  fNPFParticles = 0; fNPrimaryChildren = 0;

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

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

  // Get associations between slices and pfparticles & opt0 results
  art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, fSliceLabel);

  // Filling our neutrino hierarchy variables
  int nuID = -1; int nuSliceKey = -1;

  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;

          // We have found our neutrino!
          nuSliceKey = slice.key();
          nuID = slicePFP->Self();
          fNPFParticles = slicePFPs.size();
          fNPrimaryChildren = slicePFP->NumDaughters();

          break;
        }

      if (nuID >= 0)
        break;
    }

  if(nuID < 0)
    return;

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

Em detalhes

No arquivo de configuração, adicione a label do **track producer** e do **PFParticle producer**

Precisamos deste novo header

Vamos salvar o tamanho das tracks

Aqui vão as variáveis relacionadas a TrackLabel e PFParticleLabel que acabamos de colocar no config fcl

Inicializamos PFParticle/TrackLabel com esses labels a partir do config fcl

Em analysisConfig.fcl

```
module_type:  "AnalyseEvents"

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

Em analyzeEvents_module.cc

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

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

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

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

Criando o output

Setamos os valores salvos no vetor para cada evento em `analyzer()`

```
// Prepare variables for new event (reset counters to 0 / set default values / empty vectors)
fNPFParticles = 0;
fNPrimaryChildren = 0;
fChildTrackLengths.clear();
```

Adicionamos uma nova Branch na `TTree` usando este novo vetor em `beginJob()`

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

Os detalhes em **analyze**

Precisamos do handle de `PFParticles` para pegar as associações entre `PFParticle` -> `Track`

Vamos garantir que a `PFParticle` em questão não é o neutrino

Pegamos o vetor de `Track` objects associado a `PFParticle`.
Devemos ter apenas uma track para cada `PFParticle`.

Agora preenchemos o vetor de `Track` lengths que declaramos anteriormente

```
// Now let's look at our tracks
art::ValidHandle<std::vector<recob::PFParticle>> pfpHandle =
  e.getValidHandle<std::vector<recob::PFParticle>>(fPFParticleLabel);
art::ValidHandle<std::vector<recob::Track>> trackHandle =
  e.getValidHandle<std::vector<recob::Track>>(fTrackLabel);

art::FindManyP<recob::Track> pfpTrackAssoc(pfpHandle, e, fTrackLabel);

std::vector<art::Ptr<recob::PFParticle>> nuSlicePFPs(slicePFPAssoc.at(nuSliceKey));

// Now loop through the PFPs again to fill the track variables for the tree
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);

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

Vamos olhar nossa tracks

Igual a usar `-n -1`

Vamos rodar para todos os eventos removendo **`-n 10`** do comando:

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

Abra o arquivo de output e plote as track lengths (use `treeName->Draw("branch name")`)

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

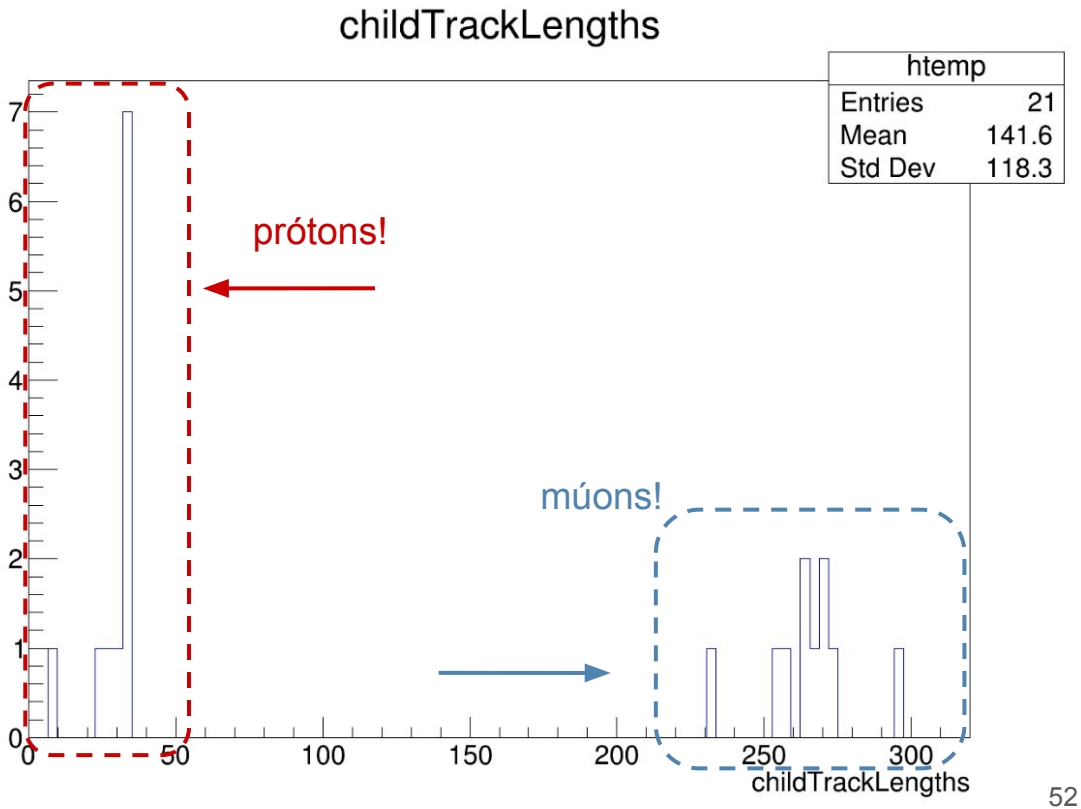
No terminal

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

No terminal do root

E aí?

Podemos ver claramente o que parecem ser prótons e múons!



6. Associações: aprofundamento

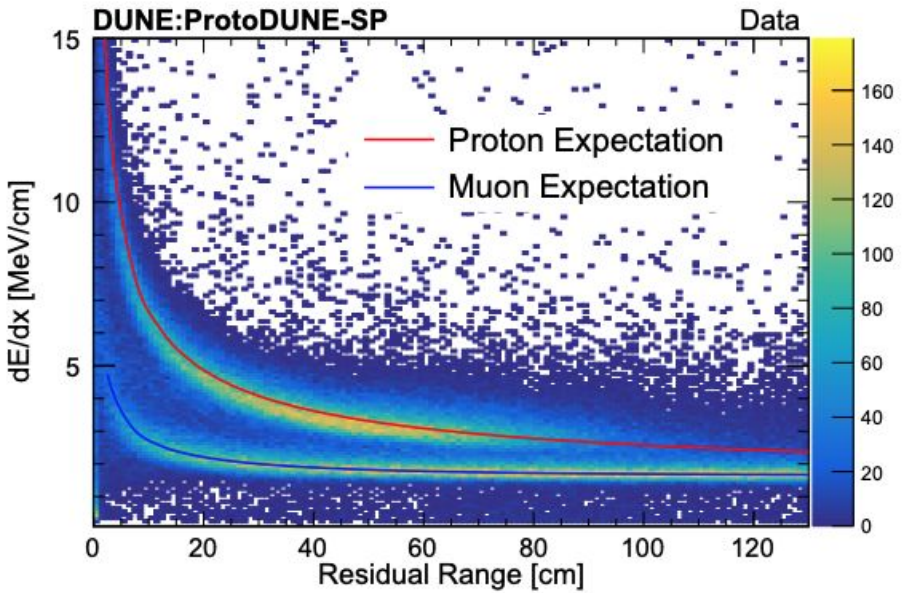
Ionização

Aqui temos um plot do ProtoDUNE-SP mostrando uma distribuição 2D de dE/dx vs. alcance residual para muons e prótons produzidos no CERN.

A distribuição teórica é mostrada através das linhas contínuas (não é um ajuste!)

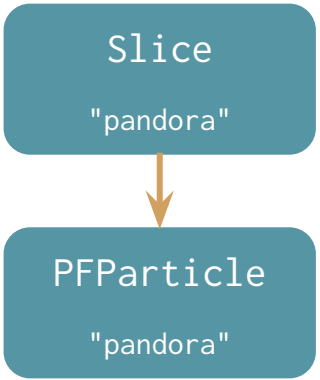
Devido à diferença de massa, temos uma ótima separação das partículas.

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



Mais associações

Anteriormente, nos atentamos à associação entre `recob::Slice`s e `recob::PFParticle`s

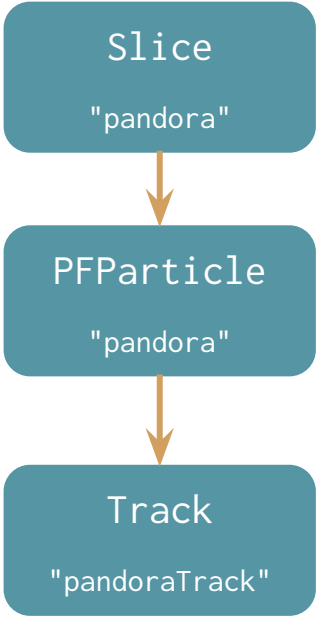


More details can be found in the doxygen entry.

Mais associações

Anteriormente, nos atentamos à associação entre `recob::Slice`s e `recob::PFParticle`s

… e depois, entre `recob::PFParticle`s e `recob::Track`s.



More details can be found in the doxygen entry.

Mais associações

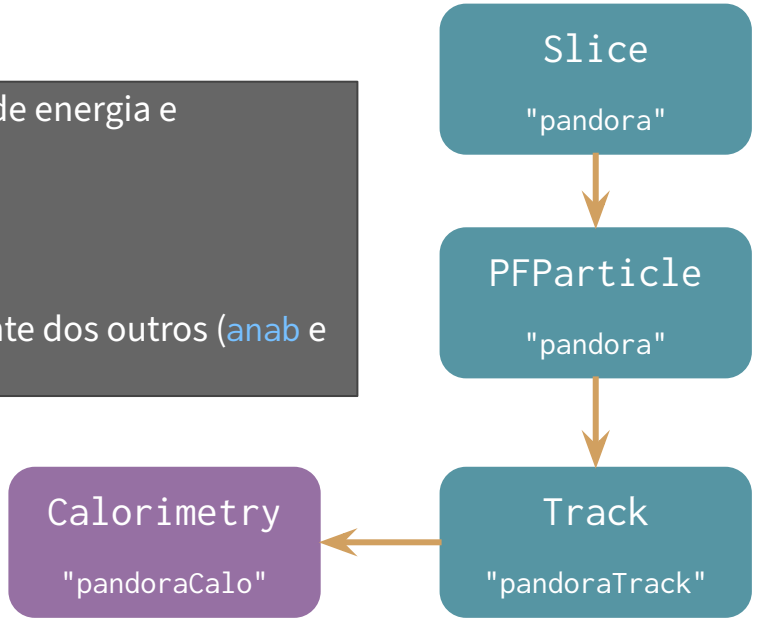
Anteriormente, nos atentamos à associação entre `recob::Slice`s e `recob::PFParticle`s

… e depois, entre `recob::PFParticle`s e `recob::Track`s.

… podemos usar ainda outra associação para descobrir a deposição de energia e reproduzir o plot do ProtoDUNE.

Agora, precisaremos do objeto `anab::Calorimetry` .

Mas preste atenção ao namespace em que ele está definido - é diferente dos outros (`anab` e não `recob`)



Mais detalhes no doxygen.

Mais associações!

Anteriormente, nos atentamos à associação entre `recob::Slice`s e `recob::PFParticle`s

… e depois, entre `recob::PFParticle`s e `recob::Track`s.

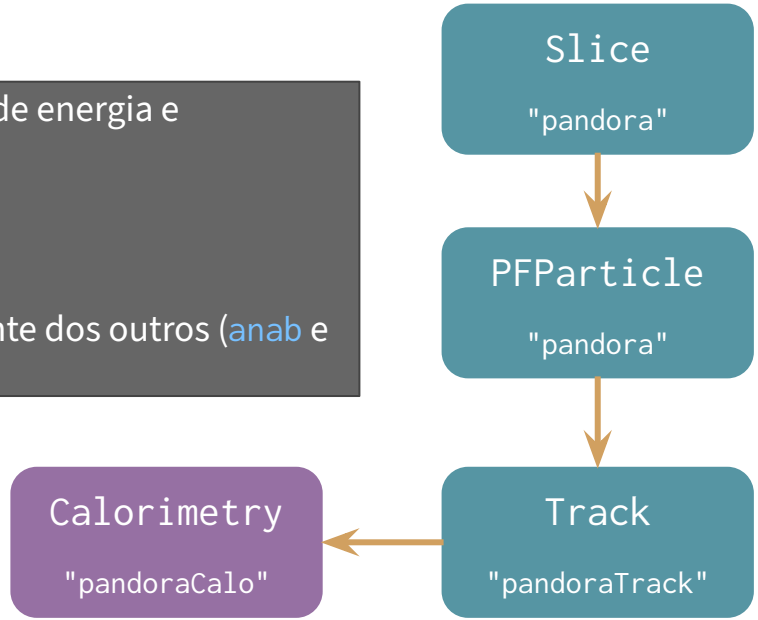
… podemos usar ainda outra associação para descobrir a deposição de energia e reproduzir o plot do ProtoDUNE.

Agora, precisaremos do objeto `anab::Calorimetry` .

Mas preste atenção ao namespace em que ele está definido - é diferente dos outros (`anab` e não `recob`)

Temos pelo menos um objeto de calorimetria para cada um dos três planos.

Os objetos contêm vetores de dQ/dx, dE/dx, Alcance Residual, etc.



Acessando a Calorimetria

Estes passos devem estar mais familiares agora:

1. Adicione o header do objeto `anab::Calorimetry` .
2. Adicione o label do módulo no seu fcl de configuração e acesse-o pelo construtor.
3. Declare e adicione Branches para novas variáveis que queira adicionar.
4. Acesse a lista de `anab::Calorimetry` de uma lista de `recob::Track` usando `art::FindManyP`.
5. Preencha sua árvore com a informação obtida de `anab::Calorimetry`.

Tente começar estes passos! Mas não se preocupe, veremos em detalhes mais adiante.

Acessando a Calorimetria

1. Adicione o header do objeto `anab::Calorimetry` .

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

2. Adicione o label do módulo no seu fcl de configuração e acesse-o pelo construtor.

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

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

```
CalorimetryLabel: "pandoracalo"
```

3. Declare e adicione Branches para novas variáveis que queira adicionar.

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

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

Acessando a Calorimetria

4. Acesse a lista de `anab::Calorimetry` de uma lista de `recob::Track` usando `art::FindManyP`.

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

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

5. Preencha sua TTree com as variáveis que tomamos do objeto `anab::Calorimetry`.

```
// 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());
       }
```

Acessando a Calorimetria

4. Acesse a lista de `anab::Calorimetry` de uma lista de `recob::Track` usando `art::FindManyP`.

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

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

5. Preencha sua TTree com as variáveis que tomamos do objeto `anab::Calorimetry`.

```
// 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());
       }
```

Lembre-se que existe um objeto para cada plano, aqui vamos considerar somente o plano coletor.

Acessando a Calorimetria

4. Acesse a lista de `anab::Calorimetry` de uma lista de `recob::Track` usando `art::FindManyP`.

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

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

5. Preencha sua TTree com as variáveis que tomamos do objeto `anab::Calorimetry`.

```
// 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());
       }
```

Podemos inserir os vetores de uma só vez com push_back()!

Hora de plotar!

Toda vez que fizermos mudanças devemos re-instalar o código. Faça isso e execute-o!

Agora você pode usar suas Branches de calorimetria e criar um histograma 2D com o ROOT.

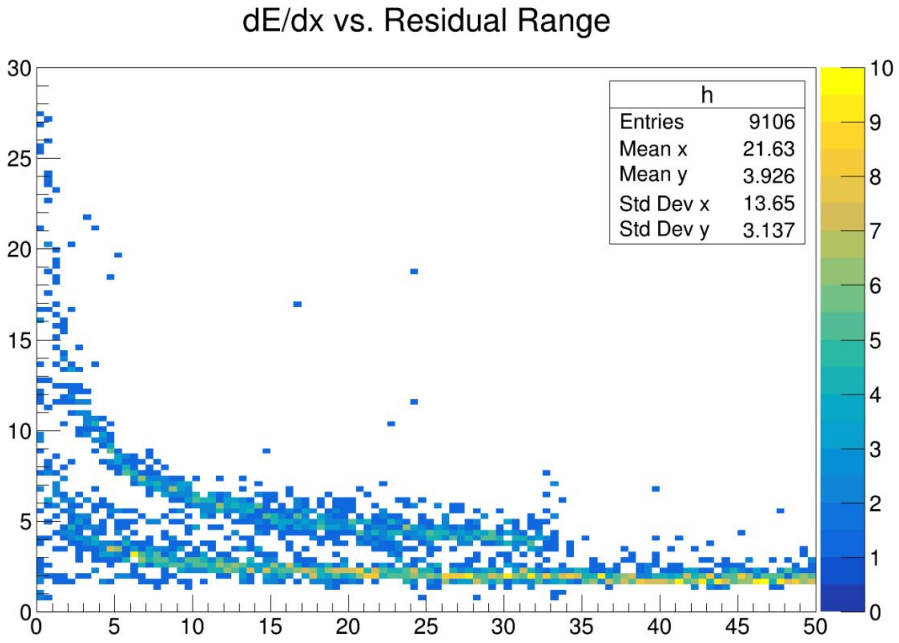
```
root[0] ana->cd()
```

```
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")
```

Você deverá encontrar algo assim:

O que você acha mais interessante nesta distribuição?



Tente mexer no estilo e legendas dos eixos do plot usando a GUI.

Você pode salvá-lo no final!

7. Um PID simplificado

Encontrando a maior track

- Como geramos um único par de próton e múon com momento definido, temos confiança de que deixarão tracks com comprimentos bastante diferentes.

- Podemos utilizar isso como uma técnica bem simples de identificação de partículas.

- Vamos fazer um loop pelas partículas filhas do neutrino e encontrar a track de maior comprimento em cada hierarquia. Faremos isso de maneira separada, antes do loop principal.

Encontrando a maior track

Vamos criar algumas variáveis para guardar qual o ID da track mais longa e qual o seu comprimento

Agora fazemos um loop nas PFPs e suas tracks associadas, da mesma forma que no loop principal da análise.

Atualizamos o valor de longestTrack se ele for menor que o da track atual

```
// 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)
       {
         // If yes, then overwrite the variables to reflect the new longest track
         longestID = track->ID();
         longestLength = track->Length();
       }
   }
```

Encontrando a maior track

No loop principal, salvamos uma variável booleana indicando se aquela track é a mais longa ou não.

```
// Was this track the one we found to be the longest earlier?
fChildTrackIsLongest.push_back(track->ID() == longestID);
```

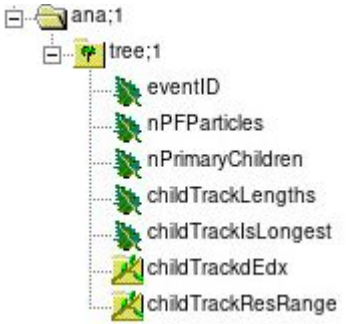
Não esqueça do passo-a-passo:

1. Declará-la
2. Inicializá-la
3. Preenchê-la (o que estamos fazendo neste slide)
4. Salvá-la em um Branch

Mais plots!

Agora sabemos quais tracks são as mais longas em cada evento. Podemos utilizar essa informação para dividir nosso plot.

Vamos abrir novamente nosso arquivo, agora fazendo uma distinção entre as duas:




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

```
root[1] TH1D *hLongest = new TH1D("hLongest", "Longest Tracks;Length;Count", 100, 0, 300)
```

```
root[2] TH1D *hOther   = new TH1D("hOther", "Other Tracks;Length;Count", 100, 0, 300)
```

Mais plots!

Vamos precisar incluir algumas opções no Draw():

```
root[3] tree->Draw("childTrackLengths>>hOther", "!childTrackIsLongest", "")
```

```
root[4] tree->Draw("childTrackLengths>>hLongest", "childTrackIsLongest", "same")
```

Vamos utilizar cores diferentes:

```
root[5] hLongest->SetLineColor(kMagenta+2)
```

Mais opções de cores aqui: https://root.cern.ch/doc/master/classTColor.html

```
root[6] hOther->SetLineColor(kOrange+2)
```

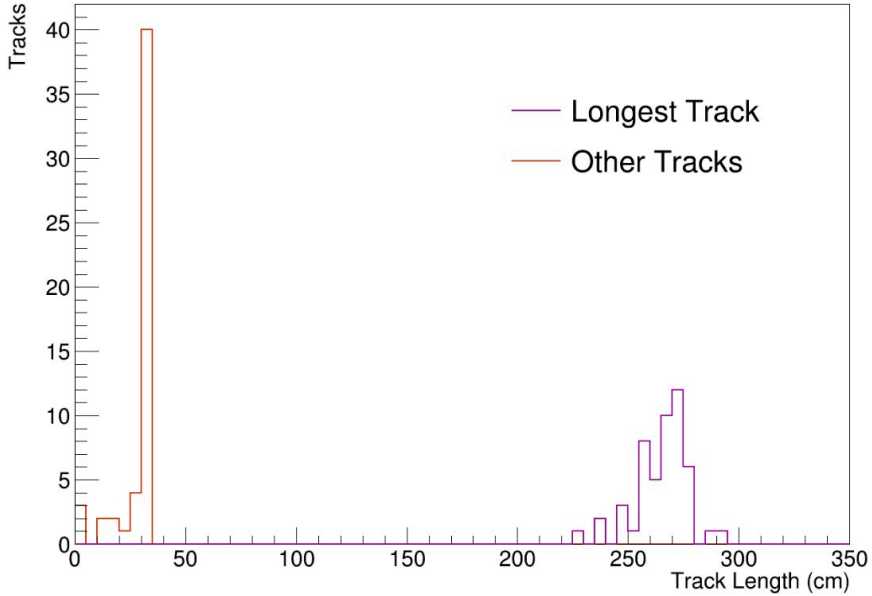
```
root[6] c1->Modified()
```

Atualizar o canvas (c1 que criamos) com nossas novas opções de cor

Comprimento

Vemos que havia dois picos na distribuição de comprimento das tracks.

Por quê?

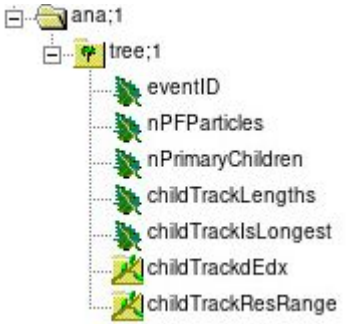


Alguns plots finais

Mais plots!

Agora sabemos quais tracks são as mais longas em cada evento. Podemos utilizar essa informação para dividir nosso plot.

Vamos abrir novamente nosso arquivo, agora fazendo uma distinção entre as duas:

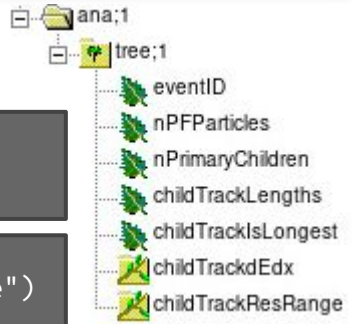



```
root[0] ana->cd()
```

```
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)
```

Mais plots!



Vamos precisar incluir algumas opções no Draw():

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

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

Vamos utilizar cores diferentes:

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

Mais opções de cores aqui: https://root.cern.ch/doc/master/classTColor.html

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

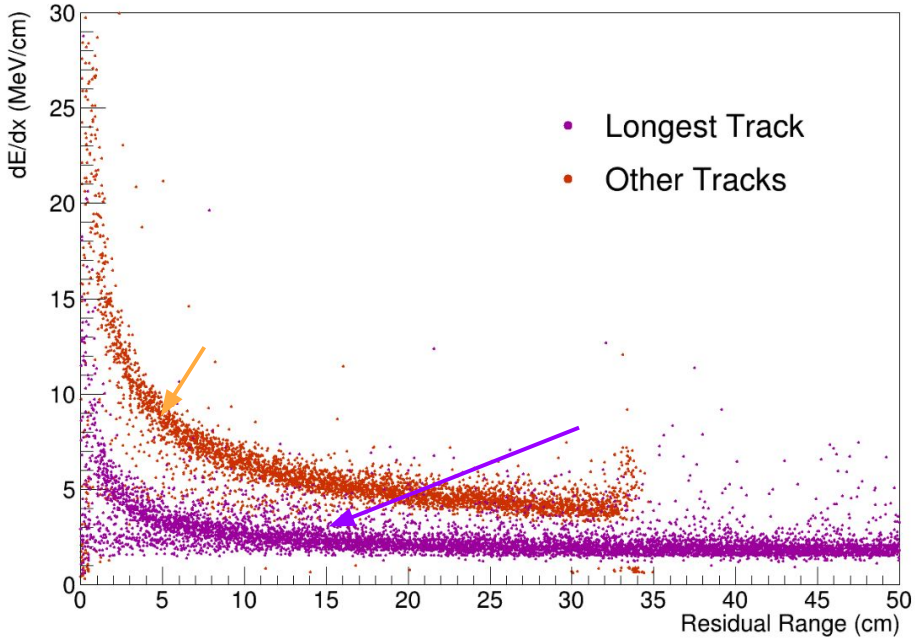
```
root[6] c1->Modified()
```

Atualizar o canvas (c1 que criamos) com nossas novas opções de cor

Deposição de energia

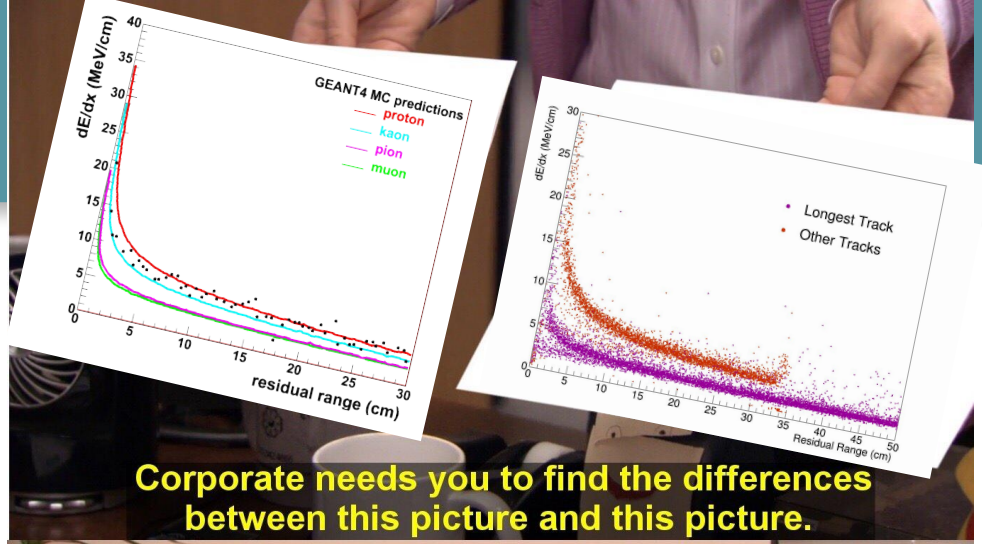
Ao plotar separadamente dE/dx vs. Alcance Residual baseado em qual track é a mais longa, vemos uma clara distinção.

Isso ocorre, pois o próton é mais altamente ionizável que o múon conforme ele se desloca no argônio.



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

Este gráfico do ArgoNeuT mostra o poder de separação teórico das distribuições médias dE/dx versus alcance residual. Os pontos pretos sobrepostos mostram uma única track contida no detector.



Extra: Outra forma de criar gráficos

Prefere Python?

Vamos tentar ler nosso arquivo de output utilizando a biblioteca uproot.

Para isso, vamos criar um ambiente virtual. Em um segundo terminal:

```
python3 -m venv my_venv
source my_venv/bin/activate
pip install -U pip
pip install matplotlib uproot awkward
```

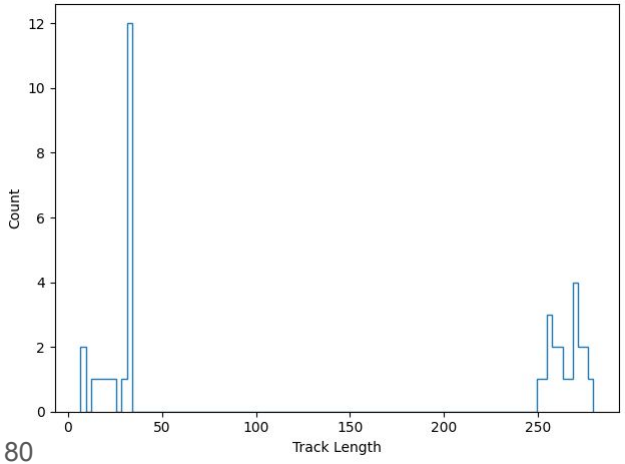
Não esqueça do container e do setup básico!

E vamos criar um arquivo .py para escrever nosso código.

Prefere Python?

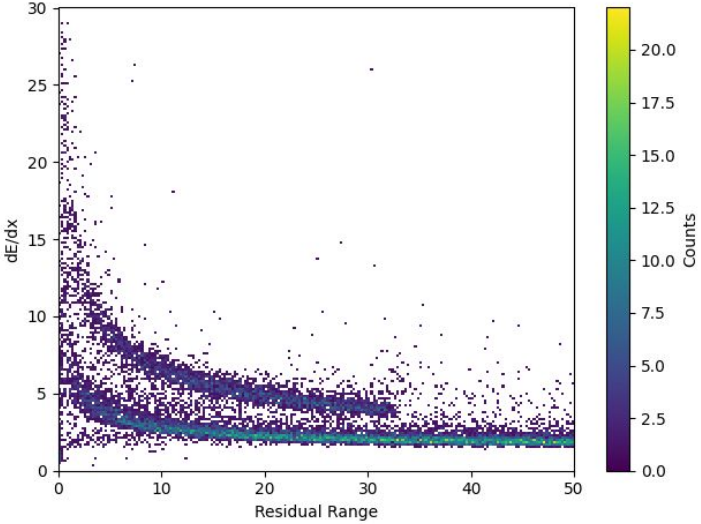
Podemos utilizar um script como este para plotar histogramas.

Tente descobrir o que cada linha faz e personalize seu gráfico!



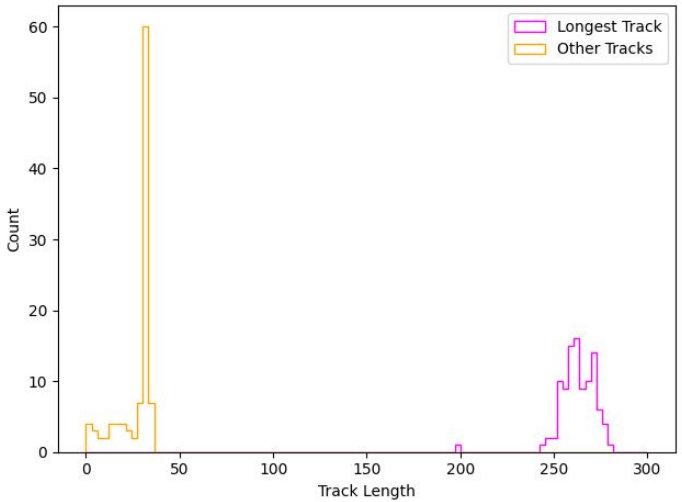

```python
import numpy as np
import matplotlib.pyplot as plt
import uproot          # For reading ROOT files
import awkward as ak   # For handling jagged arrays (variable-length lists)

# Open the ROOT file
file = uproot.open("analysisOutput.root")

# Access the TTree named "ana/tree" inside the file
tree = file["ana/tree"]

# We can read scalar branches
eventID = tree["eventID"].array(library="np")
nPFPs = tree["nPFParticles"].array(library="np")

# Read a vector branch. This is stored as a std::vector<float> in each event
trkLen = tree["childTrackLengths"].array(library="ak") # get an awkward array

# Flatten the list of lists to a single 1D array of all track lengths
trkLen = ak.flatten(trkLen)

# Plot a histogram of all track lengths
plt.hist(trkLen, bins=100, histtype='step')
plt.xlabel('Track Length')
plt.ylabel('Count')
plt.tight_layout()
plt.savefig('MyTrackLengthHistogram.png')
```

Histograma 2D




```python
import uproot
import awkward as ak
import matplotlib.pyplot as plt

file = uproot.open("analysisOutput.root")
tree = file["ana/tree"]

res_range = tree["childTrackResRange"].array(library="ak")
dedx = tree["childTrackdEdx"].array(library="ak")

res_range_flat = ak.flatten(res_range, axis=None)
dedx_flat = ak.flatten(dedx, axis=None)

res_range_np = ak.to_numpy(res_range_flat)
dedx_np = ak.to_numpy(dedx_flat)

assert len(res_range_flat) == len(dedx_flat)

cmap = plt.cm.viridis.copy()
cmap.set_under(color='white')

plt.hist2d(res_range_np, dedx_np, bins=[200, 200],
           range=[[0, 50], [0, 30]],
           cmap=cmap, vmin=1e-10)

plt.xlabel("Residual Range")
plt.ylabel("dE/dx")
plt.colorbar(label="Counts")
plt.tight_layout()
plt.savefig('MyCaloHistogram.png')
```

PID revisitado




```
import numpy as np
import matplotlib.pyplot as plt
import uproot           # For reading ROOT files
import awkward as ak    # For handling jagged arrays (variable-length lists)

file = uproot.open("nalysisOutput.root")

tree = file["ana/tree"]

trkLen = tree["childTrackLengths"].array(library="ak")
trkIsLongest = tree["childTrackIsLongest"].array(library="ak")

trkLen = ak.flatten(trkLen)
trkIsLongest = ak.flatten(trkIsLongest)

trkLongest = trkLen[trkIsLongest]
trkNotLongest = trkLen[~trkIsLongest]

plt.hist(trkLongest, bins=np.linspace(0, 300, 100), histtype='step',
color='magenta', label='Longest Track')
plt.hist(trkNotLongest, bins=np.linspace(0, 300, 100), histtype='step',
color='orange', label='Other Tracks')
plt.xlabel('Track Length')
plt.ylabel('Count')
plt.legend(loc='upper right')
plt.tight_layout()
plt.savefig('MyFirstPID.png')
```

A separação é feita aqui

Considerações Finais

Workflows

- Tutoriais com foco em ROOT via conexão VNC.
- Não tente abrir arquivos root (ou qualquer visualização) apenas com a conexão ssh.
- Frequentemente é possível estabelecer uma conexão VNC (ex.: Fermilab GPVMs).
- Outra opção é copiar os arquivos para a sua máquina local e utilizá-la para produzir a parte gráfica. Os arquivos de análise são colossalmente mais leves que os de reconstrução!

“Gabarito”

Nossa versão do código está aqui:

```
$MRB_SOURCE/duneana/duneana/CERNWorkshop/Analysis/.FinishedModule/AnalyseEvents_module.cc
```

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

```
$MRB_SOURCE/duneana/duneana/CERNWorkshop/Analysis/.FinishedModule/run_analyseEvents.fcl
```

Digite `ls -a` *para ver os diretórios e arquivos ocultos.*

Documentação e informação adicional

A documentação para cada objeto que estudamos fica aqui:

- `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
- `anab::Calorimetry` - https://code-doc.larsoft.org/docs/latest/html/classanab_1_1Calorimetry.html

Lembre-se que você pode olhar todos os objetos e seus producers correspondentes em qualquer arquivo utilizando o seu event dump:

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

Links úteis: doxygen/github

Coisas de LArSoft:

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

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

Pandora Github:

https://github.com/PandoraPFA

Específico de cada experimento:

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

Tutoriais anteriores do grupo UK (SBND-based)

Isobel Mawby & Alex Wilkinson's tutorial form 2025 is here:
https://indico.cern.ch/event/1461779/contributions/6319649/

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

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