SBN(D) CAFs: Introduction and Tutorial

Dr Thomas Jones

(Based on slides written by Dr Henry Lay)

Lancaster University

11th UK LArSoft Workshop

Thursday 29th October 2026



Outline



Click me throughout for relevant git repos!

Part 1: What are CAFs?

Part 2: CAF structure in SBN(D)

Part 3: Interactive tutorial, how to use CAFs

Outline

Join **#caf_lecture_and_tutorial** and feel free to ask questions throughout!

Part 1: What are CAFs?

Part 2: CAF structure in SBN(D)

Part 3: Interactive tutorial, how to use CAFs

Intro to CAFs

- In the analysis tutorial you have learned how to write an “Analyzer” module which produces TTrees with which you can easily analyse the data and make plots.

- CAF stands for “Common Analysis Framework”
  - CAF files are fundamentally just another example of a TTree file.
  - They are intended to contain everything necessary for high level analyses (cross-sections, oscillations, BSM searches etc).
  - No heavy products: waveforms, hits etc.
  - Designed to avoid all analysers producing their own custom trees as this is un-sustainable for the scale of SBND data and Monte Carlo.
  - Standard output from SBND productions (also used in DUNE).

Intro to CAFs

- Unlike the trees you made in your “Analyzer” tutorial the CAF files have an object driven structure.
  - We also produce “flat CAF” files which still use the structure given from the objects but as the name suggests are “flattened” into simple branches.
  - This makes the indexing of these branches very complicated - you should avoid treating these files like “standard” trees.

- CAF files can be opened and utilised in a number of ways:
  - CAFAna
  - ROOT
  - CAFPyAna
  - Uproot

Outline

Part 1: What are CAFs?

Part 2: CAF structure in SBN(D)

Part 3: Interactive tutorial, how to use CAFs

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                          TDirectoryFile
  └→ envtree                        TTree
globalTree                          TTree
recTree                             TTree
TotalPOT                             TH1D
TotalEvents                          TH1D
GenieEvtRecTree                     TTree
metadata                     TDirectoryFile
  └→ metatree                       TTree
```

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                                TDirectoryFile
  └→ envtree                              TTree
globalTree                                TTree
recTree                                   TTree
TotalPOT                                   TH1D
TotalEvents                                TH1D
GenieEvtRecTree                           TTree
metadata                       TDirectoryFile
  └→ metatree                             TTree
```

The env tree contains a dump of all the environment variables at the time of making this file

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                          TDirectoryFile
  └─→ envtree                TTree
globalTree                   TTree
recTree                      TTree
TotalPOT                     TH1D
TotalEvents                  TH1D
GenieEvtRecTree              TTree
metadata                     TDirectoryFile
  └─→ metatree               TTree
```

The global tree contains weights for each event to assess systematic uncertainties.

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                          TDirectoryFile
  └→ envtree                 TTree
globalTree                   TTree
recTree                      TTree
TotalPOT                     TH1D
TotalEvents                  TH1D
GenieEvtRecTree              TTree
metadata                     TDirectoryFile
  └→ metatree                TTree
```

The recTree is the most interesting tree. It contains the true and reconstructed “record” of the events.

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                          TDirectoryFile
  └─→ envtree                TTree
globalTree                   TTree
recTree                      TTree
TotalPOT                     TH1D
TotalEvents                  TH1D
GenieEvtRecTree              TTree
metadata                     TDirectoryFile
  └─→ metatree               TTree
```

A one bin histogram with the total POT associated with all the events in this file.

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                          TDirectoryFile
  └─→ envtree                TTree
globalTree                   TTree
recTree                      TTree
TotalPOT                     TH1D
TotalEvents                  TH1D
GenieEvtRecTree              TTree
metadata                     TDirectoryFile
  └─→ metatree               TTree
```

A one bin histogram with the total number of events in this file.

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                                TDirectoryFile
  └→ envtree                       TTree
globalTree                         TTree
recTree                            TTree
TotalPOT                           TH1D
TotalEvents                        TH1D
GenieEvtRecTree                    TTree
metadata                           TDirectoryFile
  └→ metatree                      TTree
```

The GENIE event record contains a more detailed information about the generated neutrino events (the full GENIE output), which is useful for applying cross-section systematic weights.

CAF Structure

The structure of the CAFs is defined via the objects hosted in the `sbnanaobj` repository.



```
env                          TDirectoryFile
  └─→ envtree                TTree
globalTree                   TTree
recTree                      TTree
TotalPOT                     TH1D
TotalEvents                  TH1D
GenieEvtRecTree              TTree
metadata                     TDirectoryFile
  └─→ metatree               TTree
```

metadata, similar key-value structure to the envtree, this tree contains the metadata associated with the file (code version used, production name, timestamp of creation).

CAF Structure - recTree

Throughout this tutorial, we will mainly focus on the recTree. This is the tree you should use to perform your analysis with CAFs. One entry in this tree = one event.

Lets go through the object structure of the recTree:

The StandardRecord is the top-level object. It has all true and reconstructed info, belonging to one event



Let's have a look at all the objects we have available


SRBNBinfo.h
SRBlip.h
SRBlipHitClust.h
SRCNNScore.cxx
SRCNNScore.h
SRCRTHit.cxx
SRCRTHit.h
SRCRTHitTruth.h
SRCRTPMTMatch.cxx
SRCRTPMTMatch.h
GOOD LORD, WHAT IS
HAPPENING IN THERE?

CAF Structure - recTree

Okay, so lets not go through *all* the branches

- There are over 1000
- Some of them are detector specific
- Some are data-only, MC-only
- Some of them aren't filled!

In preparation for the tutorial, let's go through some of the branches we will use.

There are a few methods to find the branches you're interested in. Personally, I like to follow the header files.

How to search

Go to the StandardRecord header. Ask yourself; what am I interested in? Slice info?

```
1   //////////////////////////////////////////////////////////////////////
2   // \file    StandardRecord.h
3   // \brief   StandardRecord defines top-level objects for
4   //          Common Analysis File trees.
5   // \author  $Author: psihas@fnal.gov
6   //////////////////////////////////////////////////////////////////////
7   #ifndef STANDARDRECORD_H
8   #define STANDARDRECORD_H
9
10  #include "sbnanaobj/StandardRecord/SRCRTHit.h"
11  #include "sbnanaobj/StandardRecord/SRCRTTrack.h"
12  #include "sbnanaobj/StandardRecord/SRCRTSpacePoint.h"
13  #include "sbnanaobj/StandardRecord/SRSBNDCRTTrack.h"
14  #include "sbnanaobj/StandardRecord/SRSBNDCRTVeto.h"
15  #include "sbnanaobj/StandardRecord/SRSBNDFrameShiftInfo.h"
16  #include "sbnanaobj/StandardRecord/SRSBNDTimingInfo.h"
17  #include "sbnanaobj/StandardRecord/SRHeader.h"
18  #include "sbnanaobj/StandardRecord/SRCRTPMTMatch.h"
19  #include "sbnanaobj/StandardRecord/SRSlice.h"
20  #include "sbnanaobj/StandardRecord/SRSliceRecoBranch.h"
21  #include "sbnanaobj/StandardRecord/SRTruthBranch.h"
22  #include "sbnanaobj/StandardRecord/SRFakeReco.h"
23  #include "sbnanaobj/StandardRecord/SROpFlash.h"
24  #include "sbnanaobj/StandardRecord/SRTrigger.h"
25  #include "sbnanaobj/StandardRecord/SRSoftwareTrigger.h"
26  #include "sbnanaobj/StandardRecord/SRInteractionDLP.h"
27  #include "sbnanaobj/StandardRecord/SRInteractionTruthDLP.h"
28  #include "sbnanaobj/StandardRecord/SRBlip.h"
```

```
#include "sbnanaobj/StandardRecord/SRSlice.h"
```

Go here!

How to search

Go to the StandardRecord header. Ask yourself; what am I interested in? Slice info?

```
1   //////////////////////////////////////////////////////////////////////
2   // \file    StandardRecord.h
3   // \brief   StandardRecord defines top-level objects for
4   //          Common Analysis File trees.
5   // \author  $Author: psihas@fnal.gov
6   //////////////////////////////////////////////////////////////////////
7   #ifndef STANDARDRECORD_H
8   #define STANDARDRECORD_H
9
10  #include "sbnanaobj/StandardRecord/SRCRTHit.h"
11  #include "sbnanaobj/StandardRecord/SRCRTTrack.h"
12  #include "sbnanaobj/StandardRecord/SRCRTSpacePoint.h"
13  #include "sbnanaobj/StandardRecord/SRSBNDCRTTrack.h"
14  #include "sbnanaobj/StandardRecord/SRSBNDCRTVeto.h"
15  #include "sbnanaobj/StandardRecord/SRSBNDFrameShiftInfo.h"
16  #include "sbnanaobj/StandardRecord/SRSBNDTimingInfo.h"
17  #include "sbnanaobj/StandardRecord/SRHeader.h"
18  #include "sbnanaobj/StandardRecord/SRCRTPMTMatch.h"
19  #include "sbnanaobj/StandardRecord/SRSlice.h"
20  #include "sbnanaobj/StandardRecord/SRSliceRecoBranch.h"
21  #include "sbnanaobj/StandardRecord/SRTruthBranch.h"
22  #include "sbnanaobj/StandardRecord/SRFakeReco.h"
23  #include "sbnanaobj/StandardRecord/SROpFlash.h"
24  #include "sbnanaobj/StandardRecord/SRTrigger.h"
25  #include "sbnanaobj/StandardRecord/SRSoftwareTrigger.h"
26  #include "sbnanaobj/StandardRecord/SRInteractionDLP.h"
27  #include "sbnanaobj/StandardRecord/SRInteractionTruthDLP.h"
28  #include "sbnanaobj/StandardRecord/SRBlip.h"
```

```
//////////////////////////////////////////////////////////////////////
#ifndef SRSLICE_H
#define SRSLICE_H

#include "sbnanaobj/StandardRecord/SRCRUMBSResult.h"
#include "sbnanaobj/StandardRecord/SRFakeReco.h"
#include "sbnanaobj/StandardRecord/SRFlashMatch.h"
#include "sbnanaobj/StandardRecord/SROpT0Finder.h"
#include "sbnanaobj/StandardRecord/SRTPCPMTBarycenterMatch.h"
#include "sbnanaobj/StandardRecord/SRCorrectedOpFlash.h"
#include "sbnanaobj/StandardRecord/SRSliceRecoBranch.h"
#include "sbnanaobj/StandardRecord/SRTrueInteraction.h"
#include "sbnanaobj/StandardRecord/SRTruthMatch.h"
#include "sbnanaobj/StandardRecord/SRVector3D.h"
#include "sbnanaobj/StandardRecord/SRNuID.h"
#include "sbnanaobj/StandardRecord/SRConstants.h"
#include "sbnanaobj/StandardRecord/SRCVNScore.h"
#include "sbnanaobj/StandardRecord/SRLightCalo.h"
#include "sbnanaobj/StandardRecord/SRNuGraphScore.h"
```

```
float    charge { kSignalingNaN };  ///< Calorimetric energy
SRVector3D charge_center;           ///< Weighted mean of hit positions in XYZ [cm]
SRVector3D charge_width;            ///< Weighted standard deviation of hit positions in XYZ [cm]
SRVector3D vertex;                  ///< Candidate neutrino vertex in local detector coordinates [cm]
```

All relevant slice variables here and also all objects that should be accessed through the slice

StandardRecord and More!

`StandardRecord rec`

StandardRecord is the top-level object from which the true and reconstructed event record is accessed.

Click here to see class header!



```
SRHeader          hdr;    ///< Header branch: run, subrun, etc.
//    SRSpill          spill;  ///< Beam spill branch: pot, beam current, etc.
SRSliceRecoBranch reco;   ///< Slice reco branch: tracks, showers, etc.
SRTruthBranch     mc;     ///< Truth branch for all interactions

int                          nslc = 0;    ///< Number of slices in list
std::vector<SRSlice>          slc;    ///< Slice branch.
int                          nfake_reco = 0; ///< Number of Fake-Reco's in list
std::vector<SRFakeReco>       fake_reco; ///< List of fake-reco slices
int                          ntrue_particles = 0; ///< Number of true particles in list
std::vector<SRTrueParticle>   true_particles; ///< True particles in spill
int                          ncrt_hits = 0; ///< Number of CRT hits in event (ICARUS)
std::vector<SRCRTHit>         crt_hits; ///< CRT hits in event (ICARUS)
int                          ncrt_tracks = 0; ///< Number of CRT tracks in event (ICARUS)
std::vector<SRCRTTrack>       crt_tracks; ///< CRT tracks in event (ICARUS)
int                          ncrt_spacepoints = 0; ///< Number of CRT spacepoints in event (SBND)
std::vector<SRCRTSpacePoint> crt_spacepoints; ///< CRT spacepoints in event (SBND)
int                          nsbnd_crt_tracks = 0; ///< Number of CRT tracks in event (SBND)
…
```

StandardRecord and More!

`StandardRecord rec` → `Vector<SRSlice> slc`

The 'slc' branches contain the information from the reconstructed slices produced by Pandora.

Click here to see class header!



```
unsigned producer { UINT_MAX };     ///< Index of the producer that produced this object.
                                      ///< In ICARUS, this is the same as the cryostat.
float    charge { kSignalingNaN };  ///< Calorimetric energy
SRVector3D charge_center;           ///< Weighted mean of hit positions in XYZ [cm]
SRVector3D charge_width;            ///< Weighted standard deviation of hit positions in XYZ [cm]
SRVector3D vertex;                  ///< Candidate neutrino vertex in local detector coordinates [cm]

SRTrueInteraction truth; //!< Truth information on the slice
SRTruthMatch tmatch; //!< Matching information between truth and reco objects

SRFlashMatch fmatch;   //!< PMT Simple flash-match for this slice of TPC charge
SRFlashMatch fmatchop;   //!< PMT Simple flash-match for this slice of TPC charge (OpFlash)
SRFlashMatch fmatchara;   //!< PMT Simple flash-match for this slice of TPC charge (XARAPUCA, SBND only)
SRFlashMatch fmatchopara;   //!< PMT Simple flash-match for this slice of TPC charge (XARAPUCA OpFlash, SBND only)

SROpT0Finder opt0;       //!< OpT0Finder (flash-match and Q->L); filled with the **highest OpT0 score** assoc. to the slice
SROpT0Finder opt0_sec;   //!< Secondary OpT0Finder (flash-match and Q->L); filled with the **second highest OpT0 score**  assoc. to the slice
```

StandardRecord and More!

`StandardRecord rec` → `Vector<SRSlice> slc` → `SRSliceRecoBranch slc`

The 'reco' branch contains access to the PFPs, Hits, and Stubs associated with that slice. The 'pfp' branches evaluate likely particle types and access track/shower objects.

Click here to see class header!



```
std::vector<SRPFP> pfp;        ///< Vector of pfps
size_t             npfp;       ///< Number of pfps

std::vector<SRHit> hit;        ///< Vector of hits
size_t            nhit;        ///< Number of hits

std::vector<SRStub> stub;      ///< Vector of stubs
size_t             nstub;      ///< Number of stubs
```

StandardRecord and More!

`StandardRecord rec` → `Vector<SRSlice> slc` → `SRSliceRecoBranch slc` → `Vector<SRPFP> pfp`

The 'reco' branch contains access to the PFPs, Hits, and Stubs associated with that slice. The 'pfp' branches evaluate likely particle types and access track/shower objects.

Click here to see class header!



```
int id;                     ///< ID of this pfp (taken from the pandora particle "ID" of this PFP)
unsigned int ndaughters;    ///< Number of daughters
std::vector<int> daughters; ///< ID's of daughters of this pfp

int parent;             ///< ID of parent particle of this pfp
bool parent_is_primary; ///< If this is a primary daughter of a neutrino/cosmic

float trackScore;  ///< The MVA score that determines how track/shower like a PFP is
SRPFOChar pfochar; ///< The MVA inputs (features) in determination of the track score (PFO Characterisation)

SRCNNScore cnnscore; // CNN scores for this PFP

SRNuGraphScore ngscore; //< NuGraph scores for this PFP

int slcID; // ID of the slice that this PFP belongs to
float t0; // T0 assigned by TPC reco, if any

SRTrack trk; // Track reconstructed from this PFP
SRShower shw; // Shower reconstructed from this PFP

SRPFPRazzled razzled; // Results from the PFP MVA
```

StandardRecord and More!

`StandardRecord rec` → `Vector<SRSlice> slc` → `SRSliceRecoBranch slc` → `Vector<SRPFP> pfp` → `SRTrack trk`

The 'trk' branches contain information from the track characterisation of this particular PFP and access to calorimetry, momentum estimates, and PID tool results.

Click here to see class header!



```
unsigned producer;    ///< Index of the producer that produced this object. In ICARUS, this is the same as the cryostat.
unsigned short npts;         ///< number of points (recob Track.NPoints)
float          len;          ///< track length [cm]
float          costh;       ///< Costh of start direction of track
float          phi;         ///< Angle of the start direction of the track in the x-y plane
SRVector3D     dir;         ///< Direction of track at start
SRVector3D     dir_end;     ///< Direction of track at end
SRVector3D     start;       ///< Start point of track
SRVector3D     end;         ///< End point of track

SRTrkChi2PID chi2pid[3]; ///< Per-plane Chi2 Particle ID
SRTrkLikelihoodPID likepid[3]; ///< Per-plane likelihood-based Particle ID variabeles (@f$ \lambda = \sum -\ln L / L_{max} @f$
where @f$\Sigma@f$ is done over hits)
SRTrackCalo calo[3]; ///< Per-plane Calorimetry information
Plane_t            bestplane;   ///< Plane index with the most hits. -1 if no calorimetry

SRTrkMCS       mcsP;
SRTrkRange     rangeP;

      …
```

StandardRecord and More!

`StandardRecord rec` → `Vector<SRSlice> slc` → `SRSliceRecoBranch slc` → `Vector<SRPFP> pfp` → `SRTrack trk` → `SRTrackCalo[3] calo`

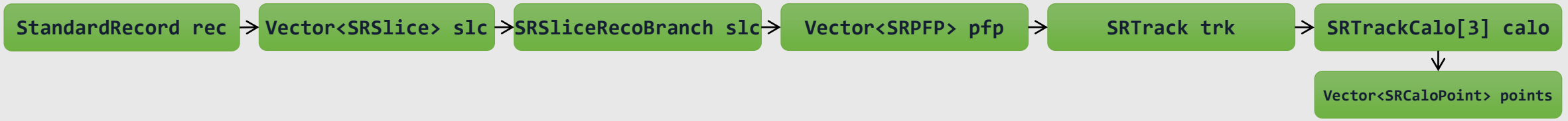
The ‘calo’ branches contain the summary track calorimetry information. They are in an array of set size 3, enumerated by plane.

Click here to see class header!



```
int              nhit;             //!< Number of hits on this plane counted in the calorimetry
float            ke;               //!< Kinetic energy deposited on this plane [GeV]
float            charge;           //!< Deposited charge as seen by wireplane (pre recombination and electric lifetime corrections) [ADC]
std::vector<SRCaloPoint> points; //!< Information saved per-point
void setDefault();
```

StandardRecord and More!



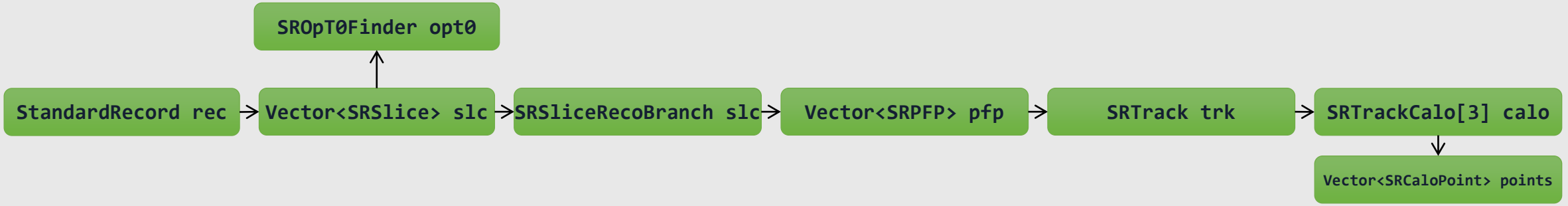
The ‘points’ branches contain the track calorimetry information point by point.

Click here to see class header!

```
float rr; //!< Residual Range [cm]
float dqdx; //!< dE/dx [MeV/cm]
float dedx; //!< dQ/dx [ADC/cm] -- pre calibration and electron lifetime correction
float pitch; //!< Track pitch [cm]
float t; //!< Time of deposition [ticks]
float efield; //! |E| [kV/cm]
float phi; //! angle (radian) between the E-field and track dir for the hit, used in the GnocchiCalorimetry module
// NOTE: flatten-caf seems to have trouble with nesting the point
// inside a SRVector3D -- so just expose x/y/z out here
float x; //!< X-Position of deposition [cm]
float y; //!< Y-Position of deposition [cm]
float z; //!< Z-Position of deposition [cm]
float integral; //!< Hit Charge Integral [ADC]
float sumadc; //!< Hit Charge SummedADC [ADC]
float width; //!< Hit width [ticks]
short mult; //!< Hit multiplicity
short wire; //!< Wire of Calo-Point
short tpc; //!< TPC of Calo-Point
short start; //!< start tick of hit
short end; //!< end tick of hit
unsigned channel; //!< Channel of Calo-Point

SRTrueCaloPoint truth; //!< Truth information
```

StandardRecord and More!



The ‘opt0’ branches contain information about the flash that best matches this slice according to the OpT0Finder algorithm.

Click here to see class header!

```
int   tpc;    // tpc that the matching was performed in
float time;    // flash-matched t0
float score;   // OpT0 score of the match; is the reciprocal of the LLH score or chi-squared score
float measPE;  // total PE of the measured flash
float hypoPE;  // total PE of the shypothetical flash
```

More Branches

We didn’t go through all the branches, so what if I wanted to know about a different branch?

There are a few methods you can use to find out

- Look into the tree structure in ROOT 
- Follow object structure in sbnanaobj
- Ask SBN CAF maintainers: Thomas Jones (Me), Gianluca Petrillo
- Git blame and ask the author

I find following object structure in sbnanaobj to be the easiest and most informative

Outline

Part 1: What are CAFs?

Part 2: CAF structure in SBN(D)

Part 3: Interactive tutorial, how to use CAFs

Tutorial Outline

Our goal is to understand how to use CAFAna to recreate the plots you made in the analysis tutorial.

This time we are working with CAF files, and not your own trees

This is an interactive tutorial, so you will have some time to work through tasks, when you see this symbol:



X mins

For some, it may take longer than specified. For some, it may be shorter. Please take advantage of the helpers and the fact that you can look at the slides in your own time

Setup



2 mins

1. Set up your development area as you have been all week

2. We will also need sbnana to provide us with CAFAna (cafe command)

   Set this up by running

```
>> setup sbnana v10_14_02_01 -q e26:prof
```

   This is the version that is consistent with the version of sbndcode we have been using in the workshop

Make your CAF file

There are a number of different CAF-making job FCLs in sbndcode. We can use the “standard” neutrino one, despite our events being particle gun.

```
>> lar -c cafmakerjob_sbnd.fcl -s /PATH/TO/YOUR/RECO2/FILE
```



2 mins

The output file name is defined in FCL

Once you have run this, you should find you have files called `<reco2_name>.caf.root` and `<reco2_name>.flat.caf.root` in your directory

If you don’t have a reco2 file you can use this prepared one:

```
>> /Workshop/sim_testsamples/reco2_withmuon_10evts.root
```

What is CAFAna?



We’re going to be using CAFAna to produce our plots. CAFAna is a wrapper around the normal ROOT framework. Why?

- Produce 1D, 2D and 3D distributions
- Make selections
- POT weight

There are two key ways of interacting with CAF files through CAFAna:

- Vars
  - Return values to be used in distributions
- Cuts
  - Return booleans for selection

Essentially, C++ functions. Can be very complex or very simple.

Proxies in CAFAna

Cuts and Vars can access the StandardRecord via two different ‘proxies’:

- `SRSpillProxy` – Cuts and Vars operate on “once per event”
- `SRSliceProxy` – Cuts and Vars operate on “once per slice”
- Anything without “Spill” in name = for slice

So we have:

- `Cut`
- `Var`
- `SpillCut`
- `SpillVar`
- `MultiVar` – Plot multiple values per proxy
- `SpillMultiVar`

Time to make a CAFAna macro!

So lets make a macro that interfaces with CAFAna and makes some plots!

The first part of our goal is to make a plot of the track lengths from our neutrino candidate primary children. This is the same plot you made in the analysis tutorial

Lets start by making a directory:

```
>> cd $MRB_SOURCE/sbndcode/sbndcode/Workshop
>> mkdir CAF
>> cd CAF
```

Time to make a CAFAna macro!



1 min

Then lets make a macro. Call it whatever you want

```
>> emacs -nw Make_CAF_Plots.C
```

Then lets add all the #includes we need:

```
#include "sbnana/CAFAna/Core/Spectrum.h"
#include "sbnana/CAFAna/Core/SpectrumLoader.h"
#include "sbnanaobj/StandardRecord/Proxy/SRProxy.h"


using namespace ana;


#include "CAFVars.h"
#include "CAFCuts.h“
#include "TCanvas.h"
#include "TH1.h"
```

Time to make a CAFAna macro!

Then lets make a macro. Call it whatever you want

```
>> emacs -nw Make_CAF_Plots.C
```

Then lets add all the #includes we need:

```
#include "sbnana/CAFAna/Core/Spectrum.h"
#include "sbnana/CAFAna/Core/SpectrumLoader.h"
#include "sbnanaobj/StandardRecord/Proxy/SRProxy.h"
```

These headers give us access to CAFAna Core utilities

```
using namespace ana;


#include "CAFVars.h"
#include "CAFCuts.h“
#include "TCanvas.h"
#include "TH1.h"
```

Time to make a CAFAna macro!

Then lets make a macro. Call it whatever you want

```
>> emacs -nw Make_CAF_Plots.C
```

Then lets add all the #includes we need:

```
#include "sbnana/CAFAna/Core/Spectrum.h"
#include "sbnana/CAFAna/Core/SpectrumLoader.h"
#include "sbnanaobj/StandardRecord/Proxy/SRProxy.h"

using namespace ana;

#include "CAFVars.h"
#include "CAFCuts.h“
#include "TCanvas.h"
#include "TH1.h"
```

ana namespace

Time to make a CAFAna macro!

Then lets make a macro. Call it whatever you want

```
>> emacs -nw Make_CAF_Plots.C
```

Then lets add all the #includes we need:

```
#include "sbnana/CAFAna/Core/Spectrum.h"
#include "sbnana/CAFAna/Core/SpectrumLoader.h"
#include "sbnanaobj/StandardRecord/Proxy/SRProxy.h"


using namespace ana;


#include "CAFVars.h"
#include "CAFCuts.h“
#include "TCanvas.h"
#include "TH1.h"
```

Header files where your Cuts and Vars will reside + ROOT headers. We will need to create the Cuts and Vars headers at some point.

Time to make a CAFAna macro!



2 mins

Now let’s add the body of our macro

```
void Make_CAF_Plots()
{
  // File stored here
  const std::string inputName = "/Workshop/CAF/reco2_withmuon_10evts.flat.caf.root";
  const TString saveDir = "./plots";


  SpectrumLoader loader(inputName);
  loader.Go();
}
```

Time to make a CAFAna macro!

Now let's add the body of our macro

```
void Make_CAF_Plots()
{
  // File stored here
  const std::string inputName = "/Workshop/CAF/reco2_withmuon_10evts.flat.caf.root";
  const TString saveDir = "./plots";


  SpectrumLoader loader(inputName);
  loader.Go();
}
```

The main function. This should have the same name as the file.

Time to make a CAFAna macro!

Now let's add the body of our macro

```
void Make_CAF_Plots()
{
  // File stored here
  const std::string inputName = "/Workshop/CAF/reco2_withmuon_10evts.flat.caf.root";
  const TString saveDir = "./plots";


  SpectrumLoader loader(inputName);
  loader.Go();
}
```

Does what it says on the tin. Change to your own file if you have one.

Time to make a CAFAna macro!

Now let's add the body of our macro

```
void Make_CAF_Plots()
{
  // File stored here
  const std::string inputName = "/Workshop/CAF/reco2_withmuon_10evts.flat.caf.root";
  const TString saveDir = "./plots";


  SpectrumLoader loader(inputName);
  loader.Go();
}
```



`SpectrumLoader` fills spectra from the dataset given to it. We've not asked for any spectra yet, so this won't do anything.

`loader.Go();` tells the spectrum loader that it's time to fill!

Time to make a CAFAna macro!



5 mins

Let’s check if this runs!

```
>> cafe -b -q Make_CAF_Plots.C
```

`Cafe` - The CAFAna command
`-b` - ROOT to run in batch mode, stops it from displaying graphics
`-q` - ROOT to quit on completion

This will compile and then run your macro. The macro will produce nothing in its current form, but it should run with no errors.

Now we can move on to making our Cuts and Vars. Let’s create the files for now:

```
>> touch CAFVars.h
>> touch CAFCuts.h
```

Time to make a CAFAna macro!



2 mins

Now let’s make our first Cut function. Place this inside your “Cuts” header.

```
const Cut kIsNeutrinoCandidateSlice([](const caf::SRSliceProxy *slc) {
    return !slc->is_clear_cosmic;
  });
```

Time to make a CAFAna macro!

Now let’s make our first Cut function. Place this inside your “Cuts” header.

```
const Cut kIsNeutrinoCandidateSlice([](const caf::SRSliceProxy *slc) {
    return !slc->is_clear_cosmic;
  });
```

This is a C++ lambda function. All your Cut and Var functions will be lambdas. This is required because CAFAna's core functions expect a callable object, allowing you to pass executable code directly into the framework.

CAFAna is designed to hide the for loops you'd normally write to iterate over a ROOT tree. When we eventually pass this Cut object to our plot, CAFAna will automatically grab every slice in our dataset, package its data into that SRSliceProxy object, and feed it into this lambda. If the lambda returns true, the slice goes into our histogram; if false, it gets rejected.

Time to make a CAFAna macro!

Now let's make our first Cut function. Place this inside your "Cuts" header.

```
const Cut kIsNeutrinoCandidateSlice([](const caf::SRSliceProxy *slc) {
    return !slc->is_clear_cosmic;
  });
```

`Cut` rather than SpillCut, as we are using `caf::SRSliceProxy`

Time to make a CAFAna macro!

Now let's make our first Cut function. Place this inside your "Cuts" header.

```
const Cut kIsNeutrinoCandidateSlice([](const caf::SRSliceProxy *slc) {
    return !slc->is_clear_cosmic;
  });
```

This is the name we are giving the cut. When we want to use it in our spectra, we will have to refer to it by this name.

The "k" prefix is a standard C++ convention denoting a constant (const) object. Meaning the rule is locked in and cannot be changed

Time to make a CAFAna macro!

Now let’s make our first Cut function. Place this inside your “Cuts” header.

```
const Cut kIsNeutrinoCandidateSlice([](const caf::SRSliceProxy *slc) {
    return !slc->is_clear_cosmic;
  });
```

The body of the function. This one is pretty simple, we are using a single field of the SRSlice object to ensure all “clear cosmics” fail this cut.



If you remember from the analyzer tutorial, you selected the slice with the highest nuscore. You may also feel free to add that in here. To access the nuscore in the same function, it is as simple as `slc->nu_score`

Time to make a CAFAna macro!



Next, we will make the Var we want to plot. Place this in your "Vars" header

```
const MultiVar kChildTrackLengths([](const caf::SRSliceProxy *slc) ->
std::vector<double> {
    std::vector<double> trackLengths;
    for(auto const& pfp : slc->reco.pfp) {
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      trackLengths.push_back(pfp.trk.len);
    }
    return trackLengths;
  });
```

While a Cut tells CAFAna whether to keep a slice, this MultiVar tells CAFAna what numbers to extract from it. CAFAna automatically feeds the slice into this lambda, takes the list of track lengths you return, and plots all of them into your histogram

Time to make a CAFAna macro!

Next, we will make the Var we want to plot. Place this in your "Vars" header

```
const MultiVar kChildTrackLengths([](const caf::SRSliceProxy *slc) ->
std::vector<double> {
    std::vector<double> trackLengths;
    for(auto const& pfp : slc->reco.pfp) {
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      trackLengths.push_back(pfp.trk.len);
    }
    return trackLengths;
  });
```

`MultiVar` as we want to plot multiple entries per slice. So the return type should be a vector

`-> std::vector<double>` Specifies that the lambda function returns a dynamically sized list (std::vector) of floating-point numbers (double)

Time to make a CAFAna macro!

Next, we will make the Var we want to plot. Place this in your "Vars" header

```
const MultiVar kChildTrackLengths([](const caf::SRSliceProxy *slc) ->
std::vector<double> {
    std::vector<double> trackLengths;
    for(auto const& pfp : slc->reco.pfp) {
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      trackLengths.push_back(pfp.trk.len);
    }
    return trackLengths;
  });
```

Loop through PFPs in the slice and consider only those whose parents were primary (and not themselves)

`continue;` breaks one iteration in a loop

Time to make a CAFAna macro!

Next, we will make the Var we want to plot. Place this in your "Vars" header

```
const MultiVar kChildTrackLengths([](const caf::SRSliceProxy *slc) ->
std::vector<double> {
    std::vector<double> trackLengths;
    for(auto const& pfp : slc->reco.pfp) {
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      trackLengths.push_back(pfp.trk.len)
    }
    return trackLengths;
  });
```

The track lengths of interest are filled into a vector, which is then returned to CAFAna

Time to make a CAFAna macro!


2 mins

Right, we're all set up to implement our plot. Lets look back to the plotting macro

```
void Make_CAF_Plots()
{
  // Your previous code here...
  SpectrumLoader loader(inputName);

  Binning trackLengthBins = Binning::Simple(70, 0, 350);
  Spectrum *sChildTrackLength = new Spectrum("sChildTrackLength", trackLengthBins,
loader, kChildTrackLengths, kNoSpillCut, kIsNeutrinoCandidateSlice);

  loader.Go();
}
```

Time to make a CAFAna macro!

Right, we're all set up to implement our plot. Lets look back to the plotting macro

```
void Make_CAF_Plots()
{
  // Your previous code here...
  SpectrumLoader loader(inputName);

  Binning trackLengthBins = Binning::Simple(70, 0, 350);
  Spectrum *sChildTrackLength = new Spectrum("sChildTrackLength", trackLengthBins,
loader, kChildTrackLengths, kNoSpillCut, kIsNeutrinoCandidateSlice);

  loader.Go();
}
```

Create a binning scheme we want to use, in this case, 70 equally divided bins between 0 and 350cm.

Note, there are other functions that support variable-length bins

Time to make a CAFAna macro!

Right, we're all set up to implement our plot. Lets look back to the plotting macro

```
void Make_CAF_Plots()
{
  // Your previous code here...
  SpectrumLoader loader(inputName);

  Binning trackLengthBins = Binning::Simple(70, 0, 350);
  Spectrum *sChildTrackLength = new Spectrum("sChildTrackLength", trackLengthBins,
loader, kChildTrackLengths, kNoSpillCut, kIsNeutrinoCandidateSlice);

  loader.Go();
}
```

Create a (1D) Spectrum object by passing: a name, a binning scheme, the loader with the relevant dataset, the Var you want to plot, any SpillCuts and any Cuts.

Note: there are lots of Spectrum constructor functions covering different dimension plots and different types of Vars.

Time to make a CAFAna macro!

Once the loader is a go, we can produce our plots



2 mins

```
void Make_CAF_Plots()
{
  // Your previous code here...
  loader.Go();

  TCanvas *cChildTrackLength = new TCanvas("cChildTrackLength", "cChildTrackLength");
  cChildTrackLength->cd();
  TH1D *hChildTrackLength = sChildTrackLength->ToTH1(sChildTrackLength->Livetime(), kLivetime);
  hChildTrackLength->SetTitle(";Track Length (cm);Primary Children");
  hChildTrackLength->Draw("histe");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.pdf");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.png");

}
```

Time to make a CAFAna macro!

Once the loader is a go, we can produce our plots!

```
void Make_CAF_Plots()
{
  // Your previous code here...
  loader.Go();

  TCanvas *cChildTrackLength = new TCanvas("cChildTrackLength", "cChildTrackLength");
  cChildTrackLength->cd();
  TH1D *hChildTrackLength = sChildTrackLength->ToTH1(sChildTrackLength->Livetime(), kLivetime);
  hChildTrackLength->SetTitle(";Track Length (cm);Primary Children");
  hChildTrackLength->Draw("histe");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.pdf");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.png");

}
```

The same as in ROOT, create a canvas to hold the plot.

Time to make a CAFAna macro!

Once the loader is a go, we can produce our plots!

```
void Make_CAF_Plots()
{
  // Your previous code here...
  loader.Go();

  TCanvas *cChildTrackLength = new TCanvas("cChildTrackLength", "cChildTrackLength");
  cChildTrackLength->cd();
  TH1D *hChildTrackLength = sChildTrackLength->ToTH1(sChildTrackLength->Livetime(), kLivetime);
  hChildTrackLength->SetTitle(";Track Length (cm);Primary Children");
  hChildTrackLength->Draw("histe");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.pdf");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.png");

}
```

Create a histogram object from our spectrum and draw it. For added style, we gave it axis titles and error bars 😎

Time to make a CAFAna macro!

Once the loader is a go, we can produce our plots!

```
void Make_CAF_Plots()
{
  // Your previous code here...
  loader.Go();

  TCanvas *cChildTrackLength = new TCanvas("cChildTrackLength", "cChildTrackLength");
  cChildTrackLength->cd();
  TH1D *hChildTrackLength = sChildTrackLength->ToTH1(sChildTrackLength->Livetime(), kLivetime);
  hChildTrackLength->SetTitle(";Track Length (cm);Primary Children");
  hChildTrackLength->Draw("histe");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.pdf");
  cChildTrackLength->SaveAs(saveDir + "/child_track_length.png");

}
```

Tell the Spectrum what exposure we want it to scale the plot to (it knows the exposure of the dataset it was given). You can pass it a POT or a Livetime (the default is POT; hence we specify kLivetime to tell it to treat the first number as a Livetime). In this scenario we scale to the original size of the sample (scale by 1).

Time to make a CAFAna macro!



3 mins

We’re now good to go and make some plots

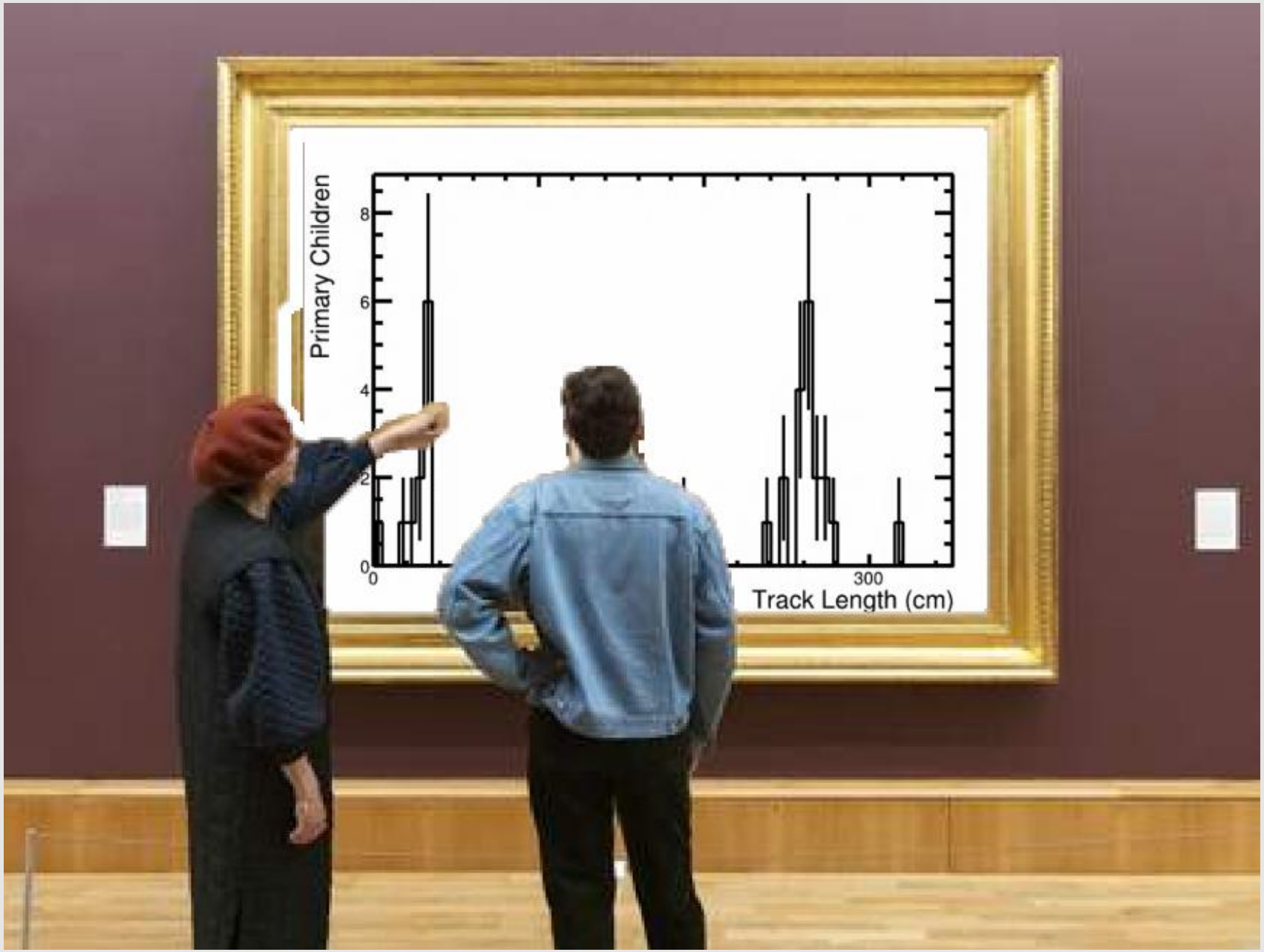
Run:

```
>> cafe -b -q Make_CAF_Plots.C
```

As before, and let’s see what happens…

A reminder that this will take a second to run

Ta-da!



Made in paint 3D

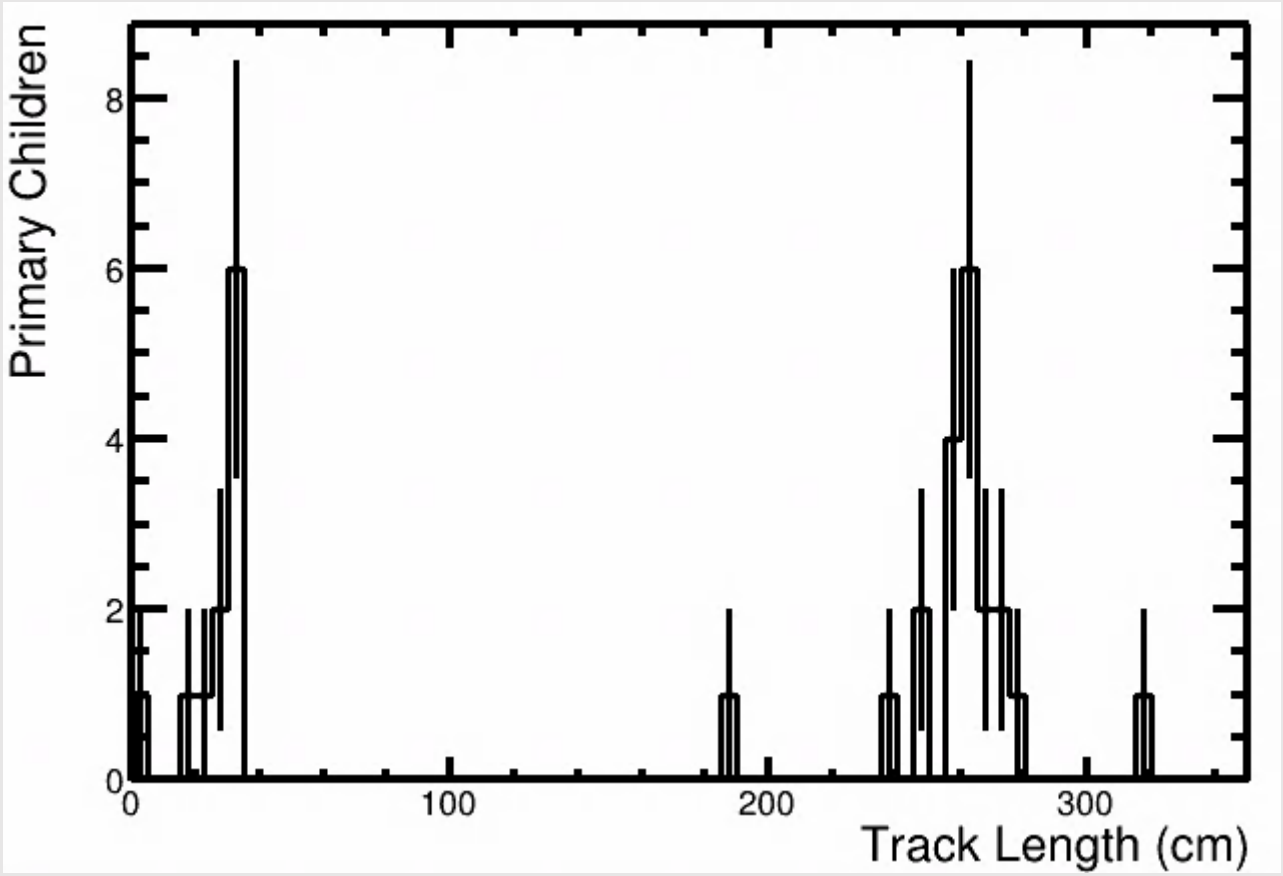
Ta-da!



1 min
To admire your plot

Very nice, we've made a 1D plot of track lengths!

This should look identical to the plot you made in the analysis tutorial ☺



A side note on style

Can create a small function using standard ROOT TStyle settings to format the plots.

For consistency, I'd recommend bundling them into "style" files that you can #include. Your collaborations may already recommend one!

```
#include "TStyle.h"
#include "TROOT.h"


void SetupStyle()
{
  gStyle->SetLineWidth(4);
  gStyle->SetHistLineWidth(3);
  gStyle->SetMarkerStyle(8);
  gStyle->SetMarkerSize(.5);
  gStyle->SetTitleOffset(.8, "y");
  gROOT->ForceStyle();
}
```

2D plots

Let's have a dig into 2D plots. We will make the 2D $\frac{dE}{dx}$ against residual range.

Our macro needs to include; 2 new binning schemes, a new spectrum and a new plotting block.

```
void Make_CAF_Plots()
{

  // Your previous code here...

  Binning dEdxBins        = Binning::Simple(180, 0, 30);
  Binning resRangeBins    = Binning::Simple(200, 0, 50);

    Spectrum *sChildTrackdEdxResRange = new Spectrum("sChildTrackdEdxResRange", loader, resRangeBins,
kChildTrackResRange, dEdxBins, kChildTrackdEdx, kNoSpillCut, kIsNeutrinoCandidateSlice);

  // …


  // 2. Child dEdx vs. residual range plot
  TCanvas *cChildTrackdEdxResRange = new TCanvas("cChildTrackdEdxResRange", "cChildTrackdEdxResRange");
  cChildTrackdEdxResRange->cd();
  TH2D *hChildTrackdEdxResRange = (TH2D*) sChildTrackdEdxResRange->ToTH2(sChildTrackdEdxResRange-
>Livetime(), kLivetime);
  hChildTrackdEdxResRange->SetTitle(";Residual Range (cm);dE/dx (MeV/cm);Primary Children");
  hChildTrackdEdxResRange->Draw("colz");
  cChildTrackdEdxResRange->SaveAs(saveDir + "/child_track_dedx_resrange.pdf");
  cChildTrackdEdxResRange->SaveAs(saveDir + "/child_track_dedx_resrange.png");


}
```

Provided code here, but now you must figure out how it fits together!

Note the key differences are in the order of arguments for the spectrum and the production of a TH2 not TH1.

Don't forget to include the TH2 ROOT header!

2D plots

We also need two variables in our Var file!



10 mins

```
const MultiVar kChildTrackdEdx([](const caf::SRSliceProxy *slc) ->
std::vector<double> {
    std::vector<double> name_of_var;
    for(auto const &pfp : slc->reco.pfp) {
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      if(pfp.trackScore < 0.5) continue;
      }
    }
    return dEdx;
  });


const MultiVar kChildTrackResRange([](const caf::SRSliceProxy *slc) ->
std::vector<double> {
    std::vector<double> name_of_var;
    for(auto const &pfp : slc->reco.pfp) {
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      }
    }
    return rr;
  });
```

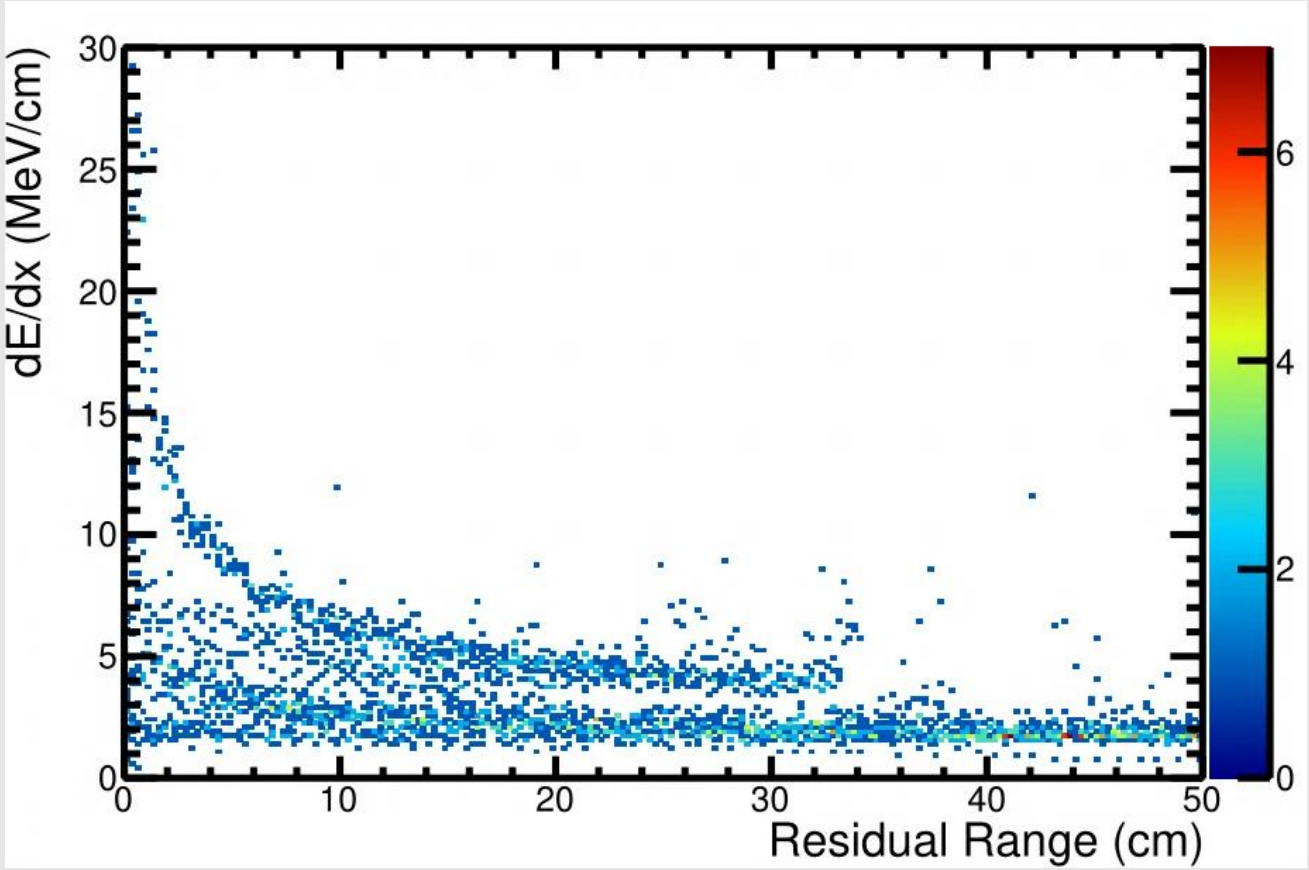
I have given the outline of the functions, but I want you to figure out how to grab the variables of interest!

Hints:

- Look at the other MultiVar function we made
- Remember back to our mapping of StandardRecord. Slide 24 may be of interest…

2D plots

You should now have the 2D $\frac{dE}{dx}$ against residual range.



Separation by track length

We now want to separate our track length and $\frac{dE}{dx}$ vs residual range by which track is longer.



5 mins

```
const Var kLongestTrack([](const caf::SRSliceProxy *slc) -> double {
    double maxLength = -1.0;
    int maxLengthID  = -1;
    int i = -1;
    for(auto const& pfp : slc->reco.pfp) {
      ++i;
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      if(pfp.trk.len > maxLength) {
  maxLength   = pfp.trk.len;
  maxLengthID = i;
      }
    }
    return maxLengthID;
  });
```

First, let's make a Var of the index of the longest track in that slice

Separation by track length

We now want to separate our track length and $\frac{dE}{dx}$ vs residual by which track is longer.



5 mins

```
const Var kChildTrackLengthLongestTrack([](const caf::SRSliceProxy *slc) -> double
{
    int maxLengthID = (int)kLongestTrack(slc);
    if(maxLengthID < 0) return -999.0;
    return slc->reco.pfp[maxLengthID].trk.len;
  });


const MultiVar kChildTrackLengthOtherTracks([](const caf::SRSliceProxy *slc) ->
std::vector<double> {
    std::vector<double> trackLengths;
    int maxLengthID = (int)kLongestTrack(slc);
    int i = -1;
    for(auto const& pfp : slc->reco.pfp) {
      ++i;
      if(!pfp.parent_is_primary || pfp.parent == -1) continue;
      if(maxLengthID == i) continue;
      trackLengths.push_back(pfp.trk.len);
    }
    return trackLengths;
  });
```

Then, we use that index to make a Var that returns the length of the longest track and a MultiVar that returns the track lengths of the other tracks

Using the examples here, you will have to make the MultiVars for dE/dx vs residual range yourself

Separation by track length





10 mins

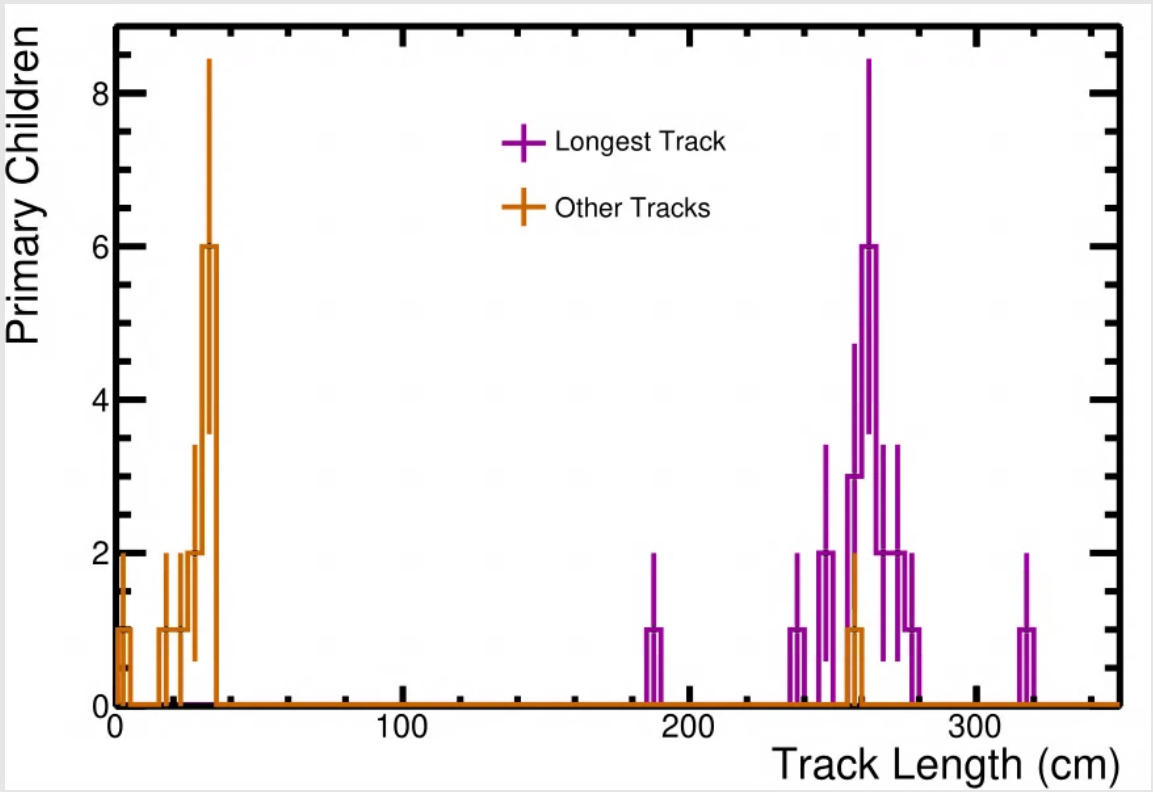
To plot these, we will then need to add the spectrums and the plotting block for all four plots. We should also make a legend

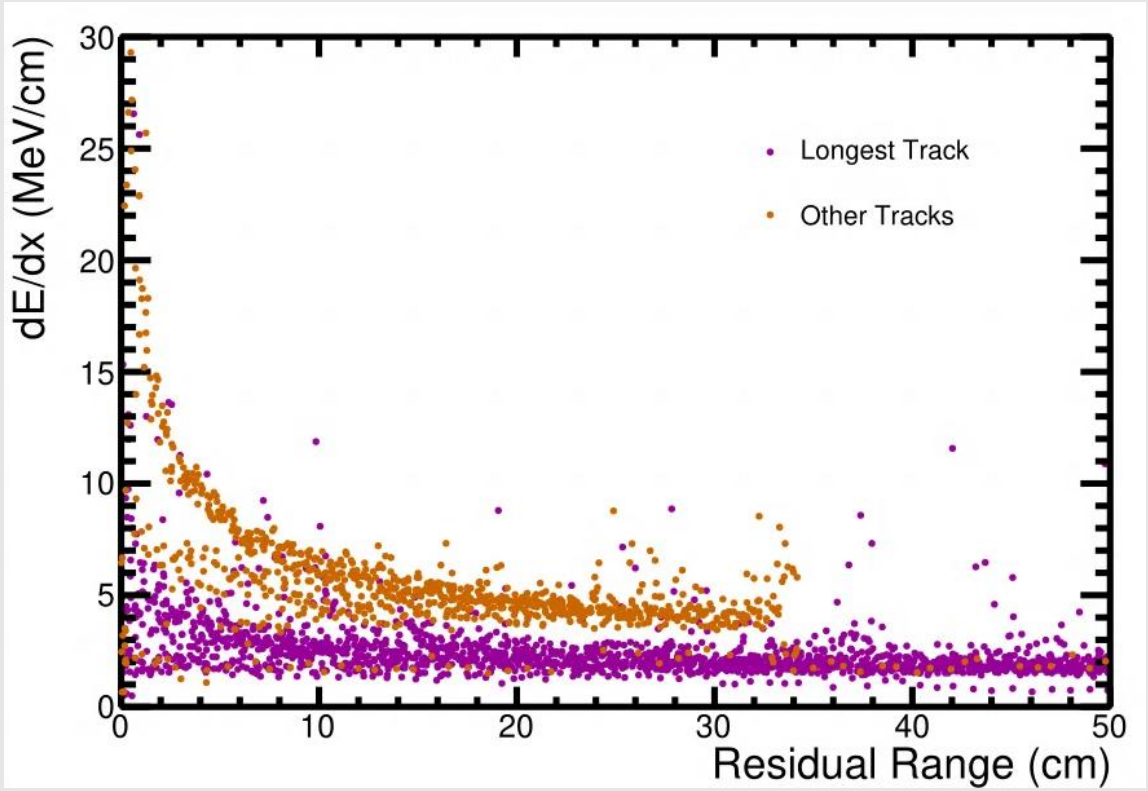
```
TLegend *lChildTrackLengthSeparated = new TLegend(.4, .65, .6, .8);
lChildTrackLengthSeparated->AddEntry(hChildTrackLengthLongestTrack, "Longest Track", "le");
lChildTrackLengthSeparated->AddEntry(hChildTrackLengthOtherTracks, "Other Tracks", "le");
```

We would also like to see the “longest track” and “Other tracks” on the same axes for each variable.

Separation by track length

You should see these nice plots now!





Last challenge: OpT0 Finder Time

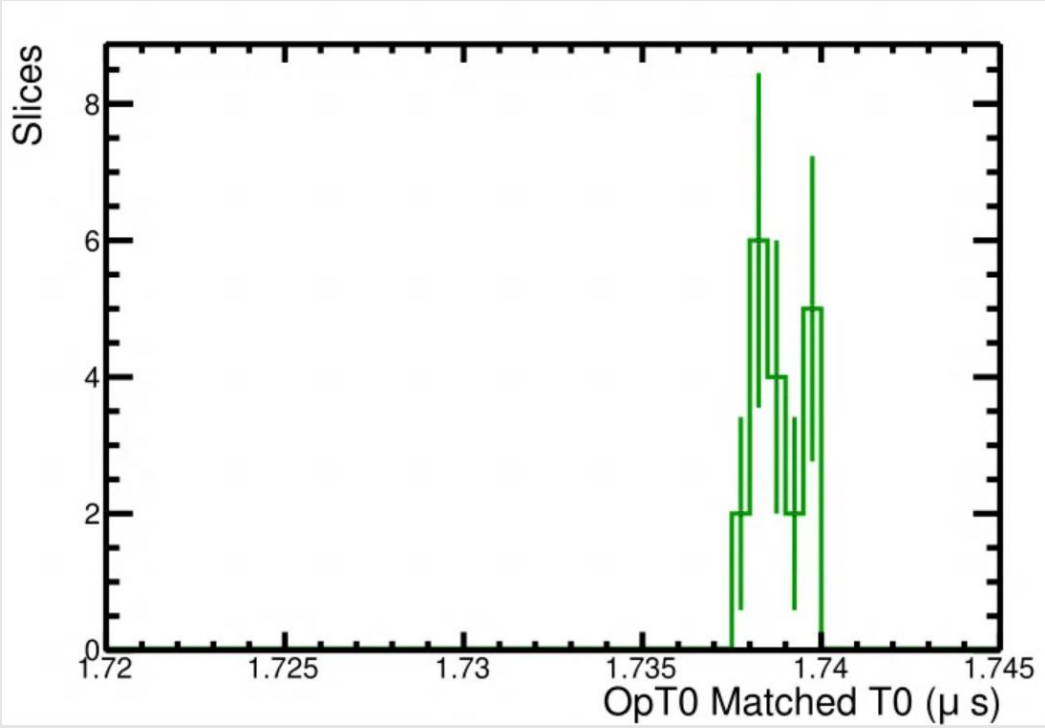


∞ mins

My last challenge to you…

Make a plot of the opt0 time, belonging to the slice. Reminder: Use the early object mapping slides. We will need a Var, a spectrum, binning and the plotting block.

You should get the following, or something similar:



More plots!

In the sbndcode workshop CAF directory, you will find the solutions to what we have gone through so far. You will also find a second set of files:

Extra_Plots.C, ExtraVars.h and ExtraCuts.h

These files are examples that use SpillVar, SpillCut and also introduce the concept of POT scaling, truth information and variable binning. Enjoy these in your own time

The End

I know you have all enjoyed this immensely. So, what have we learned?

- CAFs are tree files produced as standard outputs from SBND productions
- They have an “object” driven structure which lives in sbnanaobj
- CAFAna provides a C++ framework for analysing the CAF files.
- Sungbin has a framework for analysing SBND CAFs in python, cafpyana. See: https://github.com/sungbinoh/cafpyana/wiki

A reminder that we have a slack channel and that the solutions are in the workshop area of sbndcode, under CAF/.solutions . Feel free to use these as examples going forward.

FIN



RUN 14445, EVENT 120
July 04, 2024