SBND CAFs: Introduction and Tutorial

Dr Henry Lay
University of Sheffield

10th UK LArSoft Workshop
Wednesday 29th October 2025

RUN 14445, EVENT 120
July 04, 2024

Me!

University of Sheffield

If you haven't met me yet, I haven't been doing a very good job of TA-ing at this workshop so far!

I'm a postdoc at the University of Sheffield and was previously a PhD student at Lancaster University.

I work on SBND where I am currently one of the Reconstruction Conveners and was previously responsible for CRT Installation & Commissioning.

I am *not* a CAF expert. I have some experience using them and have attempted to turn that into a useful tutorial for you today!

Outline

Part 1: What are CAFs?

Part 2: What structure do the SBND CAFs have?

Part 3: Tutorial, how do I use the CAFs?

Outline

Part 1: What are CAFs?

Part 2: What structure do the SBND CAFs have?

Part 3: Tutorial, how do I use the CAFs?

CAF Introduction

- 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.
  - **I cannot emphasise enough that this is a key skill - producing your own trees is usually the best way to efficiently work on simulation and reconstruction developments!**

- 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).

CAF Introduction

- 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

As with all tutorials we have a slack channel #caf - please join and feel free to ask questions there!

Outline

Part 1: What are CAFs?

Part 2: What structure do the SBND CAFs have?

Part 3: Tutorial, how do I use the CAFs?

SBN(D) CAF Structure

What do I see when I open a CAF file?

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

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

SBN(D) CAF Structure

What do I see when I open a CAF file?

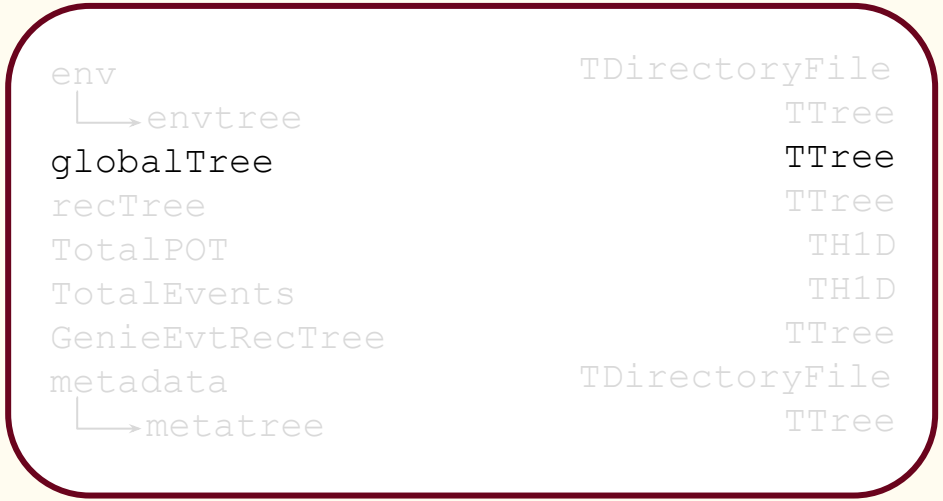
```
env                                TDirectoryFile
  └──→envtree                              TTree
globalTree                                 TTree
recTree                                    TTree
TotalPOT                                   TH1D
TotalEvents                                TH1D
GenieEvtRecTree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

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

The env tree contains a dump of the value of all the environment variables at the time of making this file… useful for some debugging purposes!

SBN(D) CAF Structure

What do I see when I open a CAF file?



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

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

SBN(D) CAF Structure

What do I see when I open a CAF file?

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

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

The recTree is the “main tree” that contains the true and reconstructed “record” of the events.

SBN(D) CAF Structure

What do I see when I open a CAF file?

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

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

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

SBN(D) CAF Structure

What do I see when I open a CAF file?

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

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

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

SBN(D) CAF Structure

What do I see when I open a CAF file?

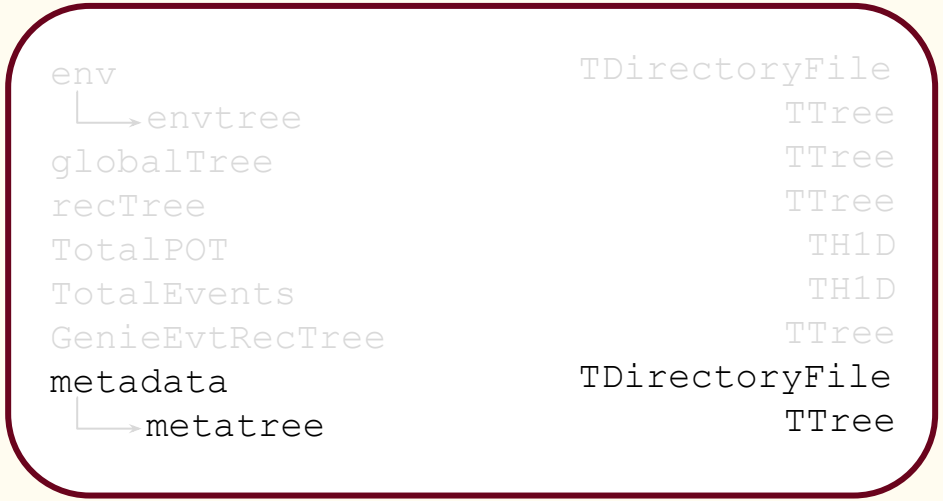
env	TDirectoryFile
└→envtree	TTree
globalTree	TTree
recTree	TTree
TotalPOT	TH1D
TotalEvents	TH1D
GenieEvtRecTree	TTree
metadata	TDirectoryFile
└→metatree	TTree

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

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.

SBN(D) CAF Structure

What do I see when I open a CAF file?



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

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).

SBN(D) CAF Structure - recTree

We will now focus on the recTree only, the main tree you would use to perform analysis with the CAFs.

I will take you through the object structure of the tree, using the object names and also the path name used in the tree.

Object

Tree Path

SBN(D) CAF Structure - recTree

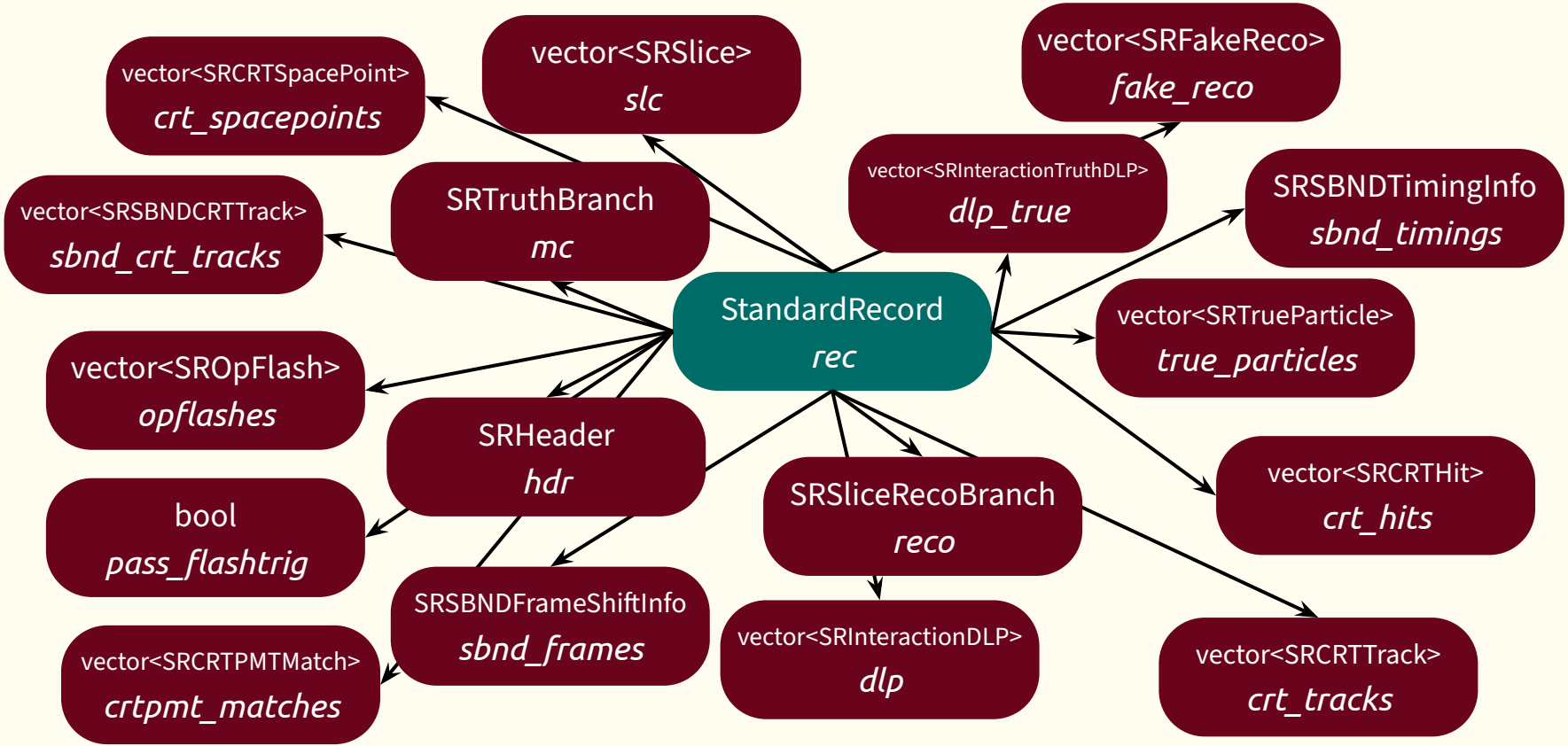
The StandardRecord is the top level object, it represents all the true and reconstructed information belonging to one “event” (by which I mean one detector readout, not one neutrino interaction).

One entry in the recTree corresponds to one event!

StandardRecord
rec

…and this is only the first level

SBN(D) CAF Structure - recTree



…and this is only the first level

SBN(D) CAF Structure - recTree

vector<SRC
crt_sp

vector<SRSlice>

vector<SRFakeReco>

vector<SRSBNDCRTT
sbnd_crt_trac

mingInfo
mings

vector<SROp
opflash

THit>
s

bool
pass_flas

vector<SRCRTP
crtpmt_m

SBN(D) CAF Structure - recTree

Object
Tree Path

Let's take a step back… We won't be going through *all* the branches for a number of reasons:

- There are well over 1000 branches.
- Some of them are SBND-only, some ICARUS-only and many are shared.
- Some of them are data-only, some are MC-only, and many are shared.
- Not all of them are filled.
- …

SBN(D) CAF Structure - recTree

Object
Tree Path

Let's take a step back… We won't be going through *all* the branches for a number of reasons:

- There are well over 1000 branches.
- Some of them are SBND-only, some ICARUS-only and many are shared.
- Some of them are data-only, some are MC-only, and many are shared.
- Not all of them are filled.
- …

And on top of all of that, I definitely do not claim to be an expert in CAFs or all their branches!

What I will do is talk you through a few common branches that relate to things you've already touched on this week. We will then use those branches to make a few plots!

Branches we care about today…

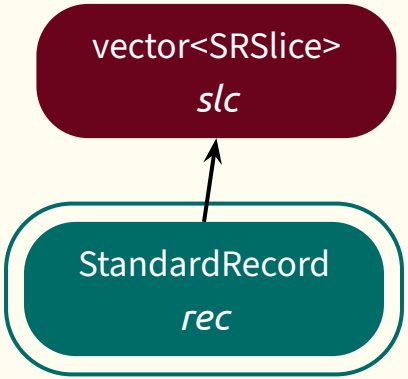
As mentioned before, the StandardRecord is the top level object from which the whole event record can be accessed.

```
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)
std::vector<SRSBNDCRTTrack>  sbnd_crt_tracks; ///< CRT tracks in event (SBND)
int                         nopflashes = 0; ///< Number of OpFlashes in spill
std::vector<SROpFlash>       opflashes; ///< List of OpFlashes in spill
int                        ncrtpmt_matches = 0; ///<Number of CRT-PMT Matches in event
std::vector<SRCRTPMTMatch> crtpmt_matches;  ///< CRT-PMT matches in event
bool pass_flashtrig = false;     ///< Whether this Record passed the Flash Trigger requirement
SRSBNDFrameShiftInfo sbnd_frames; ///< List of Frame Shift in event in unit [ns] (SBND)
SRSBNDTimingInfo sbnd_timings; ///< List of Timing Info in event in UNIX timestamp format(SBND)

int                              ndlp = 0;       ///< Number of reco DLP (ML) interactions.
std::vector<SRInteractionDLP>     dlp;        ///< Reco DLP (ML) interactions.
int                              ndlp_true = 0;  ///< Number of true DLP (ML) interactions.
std::vector<SRInteractionTruthDLP> dlp_true;   ///< True DLP (ML) interactions.
```

StandardRecord
rec

Branches we care about today…

The ‘slc’ branches contain the information from the reconstructed slices produced by pandora.



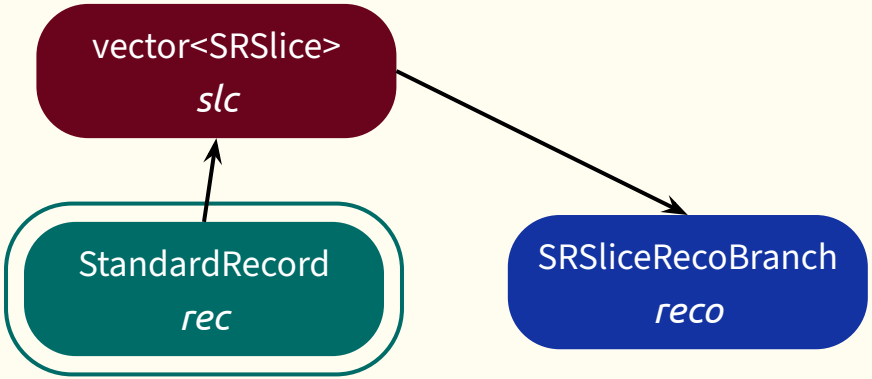

```
unsigned producer { UINT_MAX };     ///< Index of the producer that produced this object.
                                    ///< In ICARUS, this is the same as the cryostat.
float    charge { kSignalingNaN };  ///< Calorimetric energy
SRVector3D charge_center;           ///< Weighted mean of hit 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

SRTPCPMTBarycenterMatch barycenterFM; //!< Matching this slice to the OpFlash nearest to its charge center in YZ and to the triggering flash

SRCorrectedOpFlash correctedOpFlash; //!< OpFlash corrected with using tpc information matched to this slice

SRFakeReco fake_reco;

bool is_clear_cosmic { false };         //!< Whether pandora marks the slice as a "clear" cosmic
int nu_pdg           { INT_MIN };       //!< PDG assigned to the PFParticle Neutrino
float nu_score       { kSignalingNaN }; //!< Score of how neutrino-like the slice is according to pandora
float ng_filt_pass_frac { kSignalingNaN }; //!< Fraction of slice hits that pass the nugraph filter decoder
SRCRUMBSResult crumbs_result;  //!< Score of how neutrino-like the slice is according to the CRUMBS ID

SRNuID nuid; //!< Neutrino ID Features (BDT inputs) going into nu_score calculation

std::vector<size_t> primary;            //!< ID's of primary tracks and showers in slice
int                 self { INT_MIN };   //!< ID of the particle representing this slice

SRSliceRecoBranch   reco; //!< TPC reco information for the slice

SRCVNScore          cvn;  //!< Interaction type classification scores for the slice
```

Representation of artroot’s recob::Slice

Branches we care about today…

The ‘reco’ branch contains access to the PFPs, Hits (off by default) and Stubs associated with that slice.

```
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
```

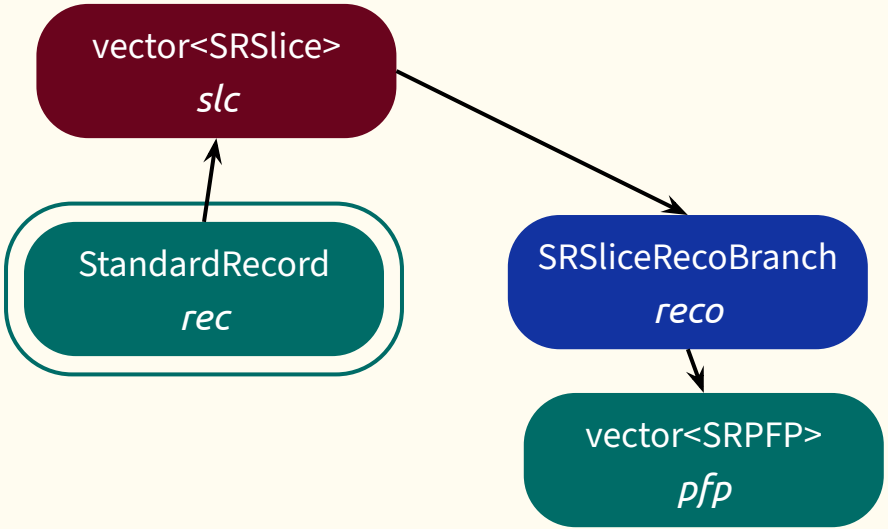


Branches we care about today…

The ‘pfp’ branches contain information about this PFP’s position within the slice’s hierarchy, evaluations of its likely particle type and access to its track and shower characterisations.

```
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
```

Representation of artroot’s recob::PFParticle

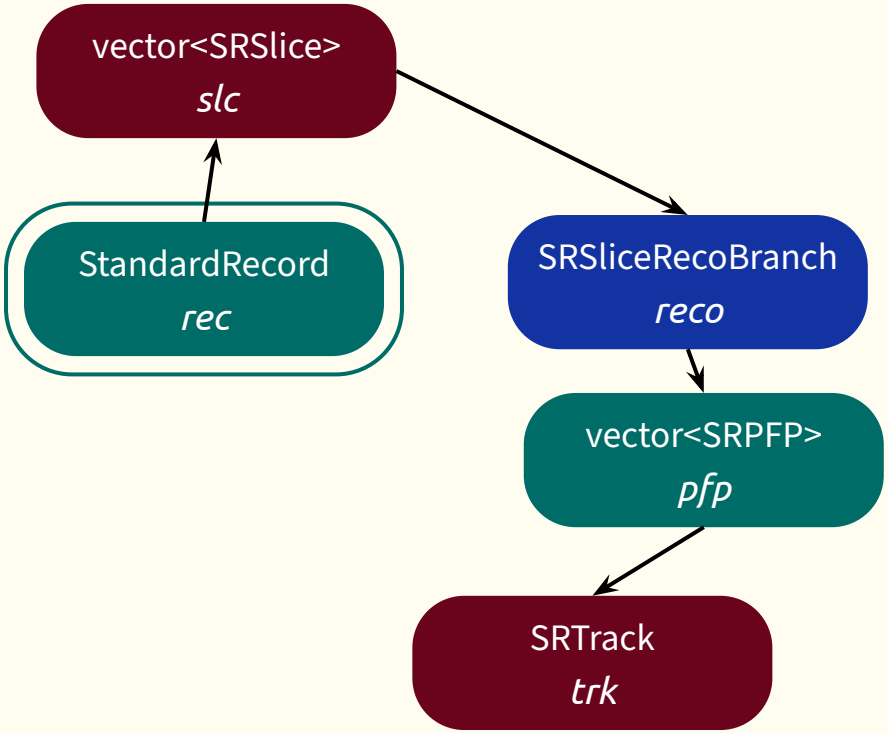


Branches we care about today…

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

```
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
SRTrackCalo calo[3]; ///< Per-plane Calorimetry information
Plane_t            bestplane;   ///< Plane index with the most hits. -1 if no calorimetry

SRTrkMCS       mcsP;
SRTrkRange     rangeP;

SRTrackTruth         truth;         ///< truth information
SRCRTHitMatch        crthit;        ///< CRT Hit match (ICARUS)
SRCRTTrackMatch      crttrack;      ///< CRT Track match (ICARUS)
SRCRTSpacePointMatch crtspacepoint; ///< CRT SpacePoint match (SBND)
SRSBNDCRTTrackMatch  crtsbndtrack;  ///< CRT Track match (SBND)

SRTrackScatterClosestApproach scatterClosestApproach; ///< Scattering variables relating to spread about interpolated track
SRTrackStoppingChi2Fit stoppingChi2Fit;               ///< Fit results from Pol0 and Exp to dEdx vs res. range
SRTrackDazzle dazzle;                                 ///< Results from the track PID MVA
```

Representation of artroot’s recob::Track

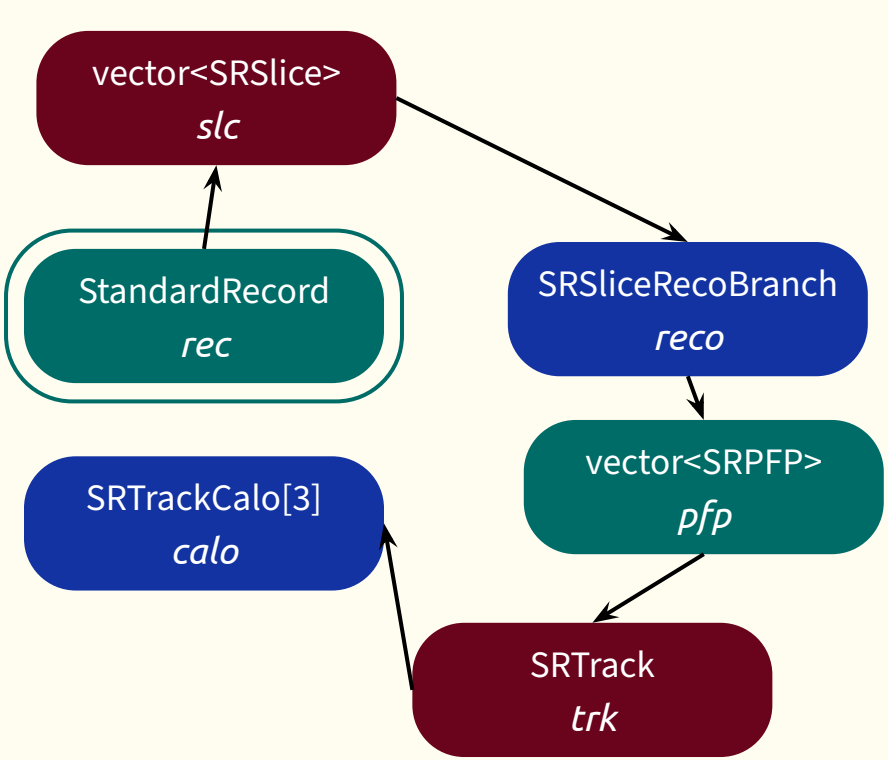


Branches we care about today…

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

```
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
```

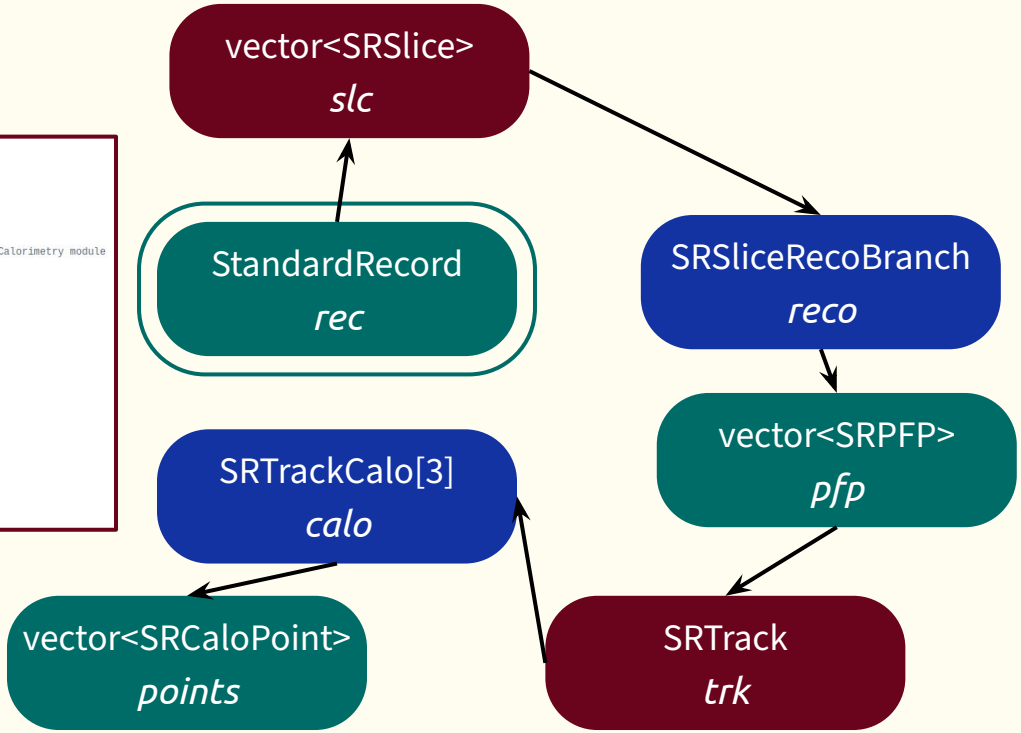
Representation of artroot’s anab::Calorimetry



Branches we care about today…

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

```
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
```

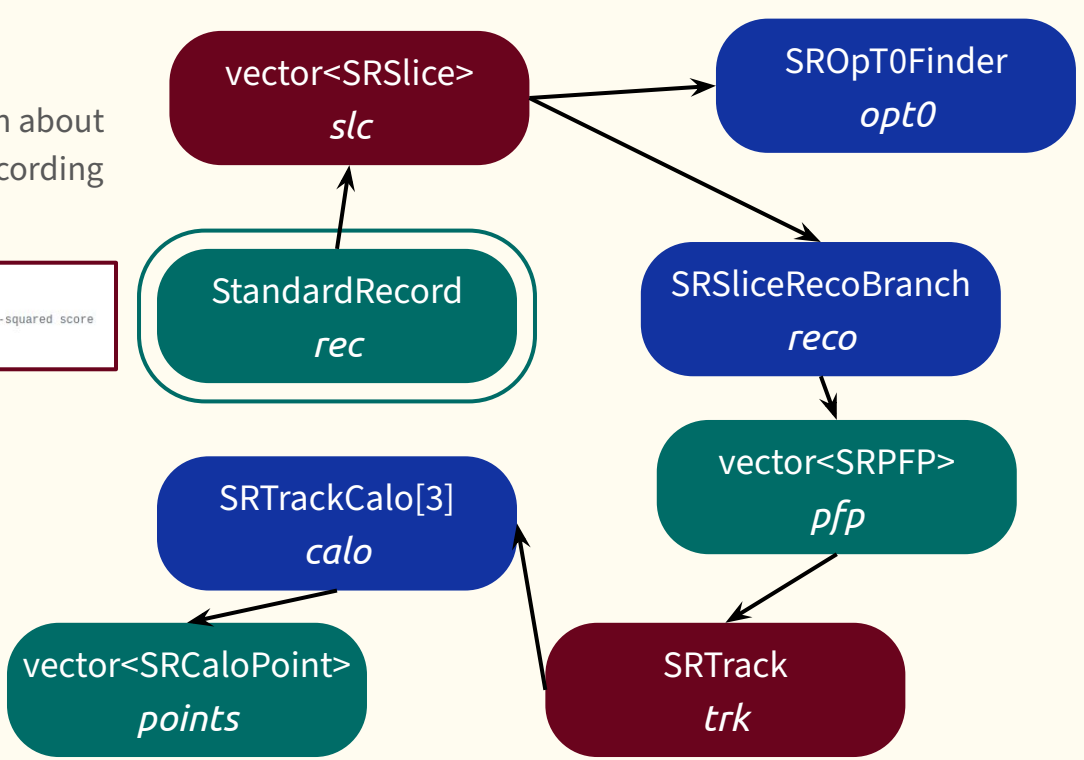


Branches we care about today…

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

```
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
```

Representation of artroot’s sbn::OpT0Finder



Some other interesting branches…

rec.slc.reco.pfp.shw

SRShower
Shower representation of PFP (recob::Shower).

rec.mc.nu

SRTrueInteraction
True neutrino interaction (simb::MCTruth).

rec.slc.reco.pfp.trk.crtspacepoint

SRCRTSpacePointMatch
CRTSpacePoint matched to a TPC track.

rec.slc.tmatch

SRTruthMatch
Slice level truth matching completeness, purity and index.

rec.slc.truth

SRTrueInteraction
True neutrino interaction matched to the slice (simb::MCTruth).

rec.hdr.pot

float
Total POT associated with the subrun.

rec.slc.reco.pfp.[trk/shw].truth

SRTrackTruth
PFP level truth matching completeness, purity and index.

rec.slc.reco.pfp.[trk/shw].truth.p

SRTrueParticle
True information about the best matched true particle.

Finding branches yourself

What about all the branches I haven't mentioned?

- Interrogating the CAF structure is more of an art than a science.
- There are a number of methods
  - Directly investigating the tree structure in interactive ROOT sessions
  - Following the object structure in sbnanaobj
  - Asking the expert in a particular aspect of the simulation/reconstruction

Finding branches yourself

What about all the branches I haven't mentioned?

- Interrogating the CAF structure is more of an art than a science.
- There are a number of methods
  - Directly investigating the tree structure in interactive ROOT sessions
  - Following the object structure in sbnanaobj
  - Asking the expert in a particular aspect of the reconstruction

I've tried to give you the tools to go down this route, each of the  symbols in previous slides is a link to the relevant object in sbnanaobj and I've given screenshots of the content of that object.

Outline

Part 1: What are CAFs?

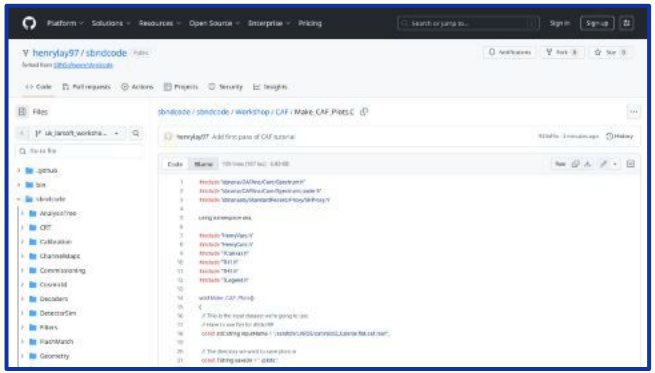
Part 2: What structure do the SBND CAFs have?

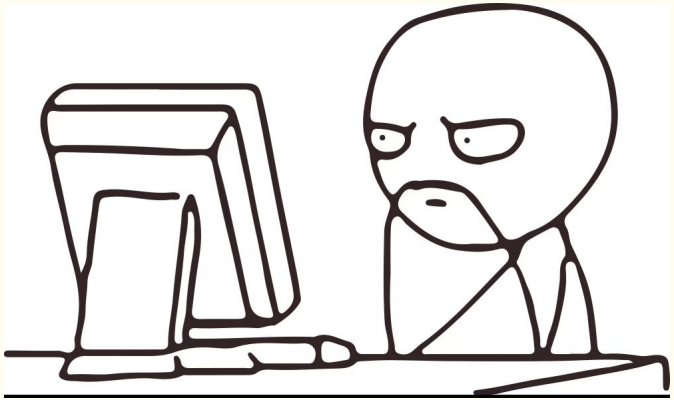
Part 3: Tutorial, how do I make and use the CAFs?

Tutorial Outline

Goal: To learn how to use CAFAna to produce plots from SBND CAF files.

Content: We'll produce the same plots that you made in the analysis tutorial but this time from the CAF files rather than your own trees. This should help you understand how the CAF structure maps to the artroot objects you've been making/using all week!





Setting Up!

1. Setup your development area as you have all week.

   We will also need sbnana to provide us with the ‘cafe’ command.

2. You can set this up by running setup sbnana v10_01_00 -q e26:prof. This is the version consistent with sbndcode v10_06_01.

This is the version used in the uk_larsoft_workshop_2025 branch!

Making your CAF file!

There are a number of different CAF-making job fcls in sbndcode ( ) they typically vary based on the type of origin (neutrino, cosmic, both), type of sample (MC, data) and whether or not you want to run systematic weight calculators or not.

We can use the “standard” neutrino one, despite our events being particle gun.

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

Note we can’t control the output file name with -T or -o, it’s form is set in the fcl file.

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: `/scratch/LAR25/reconstruction/reco2_tutorial.root`

CAFAna 101

We're going to be using CAFAna to produce our plots. CAFAna is a wrapper around the normal ROOT framework. It consists of functionality to easily produce 1D, 2D, and 3D distributions, normalise by exposure, make selections and perform oscillation fits.

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

- Vars
- Cuts

Fundamentally both are just C++ functions. Vars return values to be used in distributions whilst Cuts return booleans for selecting or rejecting. Both can call other Vars/Cuts within their implementation and can be as complex or simple as required.

CAFAna 101

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

- SRSpillProxy
- SRSliceProxy ← the ‘default’

The spill proxy allows Cuts and Vars to operate on a “once per event” basis (StandardRecord) whilst the slice proxy allows them to operate on a “once per slice” basis (SRSlice).

So we have Cut, Var, SpillCut, SpillVar, MultiVar and SpillMultiVar. The last two do what they say on the tin, allow you to plot multiple values per proxy. This is done by returning a vector of values instead of a single value.

Baby’s First CAFAna Macro

So we know the boring “on paper” CAFAna basics, how does this actually look in reality though?

Baby’s First CAFAna Macro

So we know the boring “on paper” CAFAna basics, how does this actually look in reality though?

Make yourself a directory in your workshop area:

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

Baby’s First CAFAna Macro

In that directory we are going to make our macro. I’m calling mine Make_CAF_Plots.C and opening it with emacs.

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

Baby’s First CAFAna Macro

In that directory we are going to make our macro. I’m calling mine Make_CAF_Plots.C and opening it with emacs.

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

Reminder, the first plot we’re planning on making is of all the track lengths from our neutrino candidate primary children…

Baby's First CAFAna Macro

In that directory we are going to make our macro. I'm calling mine Make_CAF_Plots.C and opening it with emacs.

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

First let's add all the #includes we're going to need for our first plot…

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

using namespace ana;

#include "HenryVars.h"
#include "HenryCuts.h"
#include "TCanvas.h"
#include "TH1.h"
```

Baby's First CAFAna Macro

In that directory we are going to make our macro. I'm calling mine Make_CAF_Plots.C and opening it with emacs.

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

First let's add all the #includes we're going to need for our first plot…

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

using namespace ana;

#include "HenryVars.h"
#include "HenryCuts.h"
#include "TCanvas.h"
#include "TH1.h"
```

These headers give us access to the CAFAna core utilities we want to use.

Baby's First CAFAna Macro

In that directory we are going to make our macro. I'm calling mine Make_CAF_Plots.C and opening it with emacs.

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

First let's add all the #includes we're going to need for our first plot…

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

using namespace ana;

#include "HenryVars.h"
#include "HenryCuts.h"
#include "TCanvas.h"
#include "TH1.h"
```

This is the namespace all the useful CAF objects live in. By declaring it here we don't have to write it out everytime, both in subsequent header files and the main body.

Baby's First CAFAna Macro

In that directory we are going to make our macro. I'm calling mine Make_CAF_Plots.C and opening it with emacs.

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

First let's add all the #includes we're going to need for our first plot…

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

using namespace ana;

#include "HenryVars.h"
#include "HenryCuts.h"
#include "TCanvas.h"
#include "TH1.h"
```

Header files where your personal Cuts and Vars are going to live. You can call them what you like.

Don't forget to create them (empty for now)

```
touch HenryVars.h
touch HenryCuts.h
```

Baby's First CAFAna Macro

In that directory we are going to make our macro. I'm calling mine Make_CAF_Plots.C and opening it with emacs.

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

First let's add all the #includes we're going to need for our first plot…

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

using namespace ana;

#include "HenryVars.h"
#include "HenryCuts.h"
#include "TCanvas.h"
#include "TH1.h"
```

ROOT plotting objects we are going to use.

Baby's First CAFAna Macro

Now let's add a skeleton body to our macro…

```
void Make_CAF_Plots()
{
  // This is the input dataset we're going to use
  const std::string inputName = "/scratch/LAR25/caf/reco2_tutorial.flat.caf.root";

  // The directory we want to save plots in
  const TString saveDir = "./plots";

  // This object takes the dataset and fills all the 'spectra' you request from it
  SpectrumLoader loader(inputName);

  // Tell the loader we've declared all our spectra, time to fill them!
  loader.Go();
}
```

Baby's First CAFAna Macro

Now let's add a skeleton body to our macro…

```
void Make_CAF_Plots()
{
  // This is the input dataset we're going to use
  const std::string inputName = "/scratch/LAR25/caf/reco2_tutorial.flat.caf.root";

  // The directory we want to save plots in
  const TString saveDir = "./plots";

  // This object takes the dataset and fills all the 'spectra' you request from it
  SpectrumLoader loader(inputName);

  // Tell the loader we've declared all our spectra, time to fill them!
  loader.Go();
}
```

Like with ROOT, the main function name should match the file name.

Baby's First CAFAna Macro

Now let's add a skeleton body to our macro…

```
void Make_CAF_Plots()
{
  // This is the input dataset we're going to use
  const std::string inputName = "/scratch/LAR25/caf/reco2_tutorial.flat.caf.root";

  // The directory we want to save plots in
  const TString saveDir = "./plots";

  // This object takes the dataset and fills all the 'spectra' you request from it
  SpectrumLoader loader(inputName);

  // Tell the loader we've declared all our spectra, time to fill them!
  loader.Go();
}
```

This points to the pre-prepared file I made, you can use this or point to your own!
The CAFAna functionality also supports using wildcards or samweb queries in this field.

Baby's First CAFAna Macro

Now let's add a skeleton body to our macro…

```
void Make_CAF_Plots()
{
  // This is the input dataset we're going to use
  const std::string inputName = "/scratch/LAR25/caf/reco2_tutorial.flat.caf.root";

  // The directory we want to save plots in
  const TString saveDir = "./plots";

  // This object takes the dataset and fills all the 'spectra' you request from it
  SpectrumLoader loader(inputName);

  // Tell the loader we've declared all our spectra, time to fill them!
  loader.Go();
}
```

Choose somewhere to save the plots, please make sure the directory exists!

Baby’s First CAFAna Macro

Now let’s add a skeleton body to our macro…

```
void Make_CAF_Plots()
{
  // This is the input dataset we're going to use
  const std::string inputName = "/scratch/LAR25/caf/reco2_tutorial.flat.caf.root";

  // The directory we want to save plots in
  const TString saveDir = "./plots";

  // This object takes the dataset and fills all the 'spectra' you request from it
  SpectrumLoader loader(inputName);

  // Tell the loader we've declared all our spectra, time to fill them!
  loader.Go();
}
```

The SpectrumLoader is a key object in CAFAna, it fills spectra from the dataset given to it. In our skeleton function we’ve not asked for any spectra yet!

Baby’s First CAFAna Macro

Let’s just check this macro runs!

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

‘cafe’ is the CAFAna command (a wrapper on the ‘root’ command)
-b tells ROOT to run in batch mode (no graphics)
-q tells ROOT to quit on completion of the macro

Remember we haven’t really told the macro to do anything yet, so we’re not expecting any plots!

Baby’s First CAFAna Macro

Time to make a first Cut function. This should be placed inside your ‘Cuts’ header. The purpose of this first cut is to ensure we only look at Pandora’s neutrino candidate slices.

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

Baby’s First CAFAna Macro

Time to make a first Cut function. This should be placed inside your ‘Cuts’ header. The purpose of this first cut is to ensure we only look at Pandora’s neutrino candidate slices.

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

As we learned earlier, Cut is used on a SliceProxy (with SpillCut being used on SpillProxy).

Baby’s First CAFAna Macro

Time to make a first Cut function. This should be placed inside your ‘Cuts’ header. The purpose of this first cut is to ensure we only look at Pandora’s neutrino candidate slices.

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

This is the name we will use to enact this cut in our spectra.

Baby’s First CAFAna Macro

Time to make a first Cut function. This should be placed inside your ‘Cuts’ header. The purpose of this first cut is to ensure we only look at Pandora’s neutrino candidate slices.

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

Finally, and most importantly, the body. In this case the body is incredibly simple, we just use a single field of the SRSlice object () in order to ensure all “clear cosmics” fail this cut.

Baby's First CAFAna Macro

Next we will make the Var we want to plot. Place this in your 'Vars' header. We want to plot the track lengths for children of the neutrino candidate.

```
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;
  });
```

We are using a MultiVar because we want to plot multiple entries per slice. As such, the return type must be a vector.

Baby's First CAFAna Macro

Next we will make the Var we want to plot. Place this in your 'Vars' header. We want to plot the track lengths for children of the neutrino candidate.

```
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;
  });
```

We loop through all the PFPs in the slice, and we only consider those whose parents were the primary (we also have to exclude the primary itself).

Baby's First CAFAna Macro

Next we will make the Var we want to plot. Place this in your 'Vars' header. We want to plot the track lengths for children of the neutrino candidate.

```
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.

Baby's First CAFAna Macro

Next we will make the Var we want to plot. Place this in your 'Vars' header. We want to plot the track lengths for children of the neutrino candidate.

```
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;
  });
```

Do you recognise the branch names from our map earlier?

Baby's First CAFAna Macro

Let's move back to the macro file and implement our plot. We now add a couple of lines between creating our SpectrumLoader and setting it off.

```
// This object takes the dataset and fills all the 'spectra' you request from it
SpectrumLoader loader(inputName);

// Binning schemes we want to use for our plots
Binning trackLengthBins = Binning::Simple(70, 0, 350);

// The 'spectra' we want to create from our dataset
Spectrum *sChildTrackLength = new Spectrum("sChildTrackLength", trackLengthBins, loader, kChildTrackLengths,
                                           kNoSpillCut, kIsNeutrinoCandidateSlice);

// Tell the loader we've declared all our spectra, time to fill them!
loader.Go();
```

Baby's First CAFAna Macro

Let's move back to the macro file and implement our plot. We now add a couple of lines between creating our SpectrumLoader and setting it off.

```
// This object takes the dataset and fills all the 'spectra' you request from it
SpectrumLoader loader(inputName);

// Binning schemes we want to use for our plots
Binning trackLengthBins = Binning::Simple(70, 0, 350);

// The 'spectra' we want to create from our dataset
Spectrum *sChildTrackLength = new Spectrum("sChildTrackLength", trackLengthBins, loader, kChildTrackLengths,
                                           kNoSpillCut, kIsNeutrinoCandidateSlice);

// Tell the loader we've declared all our spectra, time to fill them!
loader.Go();
```

Create a binning scheme we want to use, in this case 70 equally divided bins between 0 and 350cm. There are other functions (not 'Simple') that support variable length bins ().

Baby's First CAFAna Macro

Let's move back to the macro file and implement our plot. We now add a couple of lines between creating our SpectrumLoader and setting it off.

```
// This object takes the dataset and fills all the 'spectra' you request from it
SpectrumLoader loader(inputName);

// Binning schemes we want to use for our plots
Binning trackLengthBins = Binning::Simple(70, 0, 350);

// The 'spectra' we want to create from our dataset
Spectrum *sChildTrackLength = new Spectrum("sChildTrackLength", trackLengthBins, loader, kChildTrackLengths,
                                           kNoSpillCut, kIsNeutrinoCandidateSlice);

// Tell the loader we've declared all our spectra, time to fill them!
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. There are lots of Spectrum constructor functions () covering different dimension plots and different types of Vars.

Baby's First CAFAna Macro

Finally, once the loader has run, we can produce our plots.

```
// Tell the loader we've declared all our spectra, time to fill them!
loader.Go();

// Child track length plot
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");
}
```

Baby's First CAFAna Macro

Finally, once the loader has run, we can produce our plots.

```
// Tell the loader we've declared all our spectra, time to fill them!
loader.Go();

// Child track length plot
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 canvas to host the plot.

Baby's First CAFAna Macro

Finally, once the loader has run, we can produce our plots.

```
// Tell the loader we've declared all our spectra, time to fill them!
loader.Go();

// Child track length plot
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 then draw it! We give it axis titles and then draw it with (statistical) error bars.

Baby's First CAFAna Macro

Finally, once the loader has run, we can produce our plots.

```
// Tell the loader we've declared all our spectra, time to fill them!
loader.Go();

// Child track length plot
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");
```

Crucially we have to 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). I have 'hacked' it slightly to ensure in this scenario we scale to the original size of the sample (scale by 1).

Baby's First CAFAna Macro

Finally, once the loader has run, we can produce our plots.

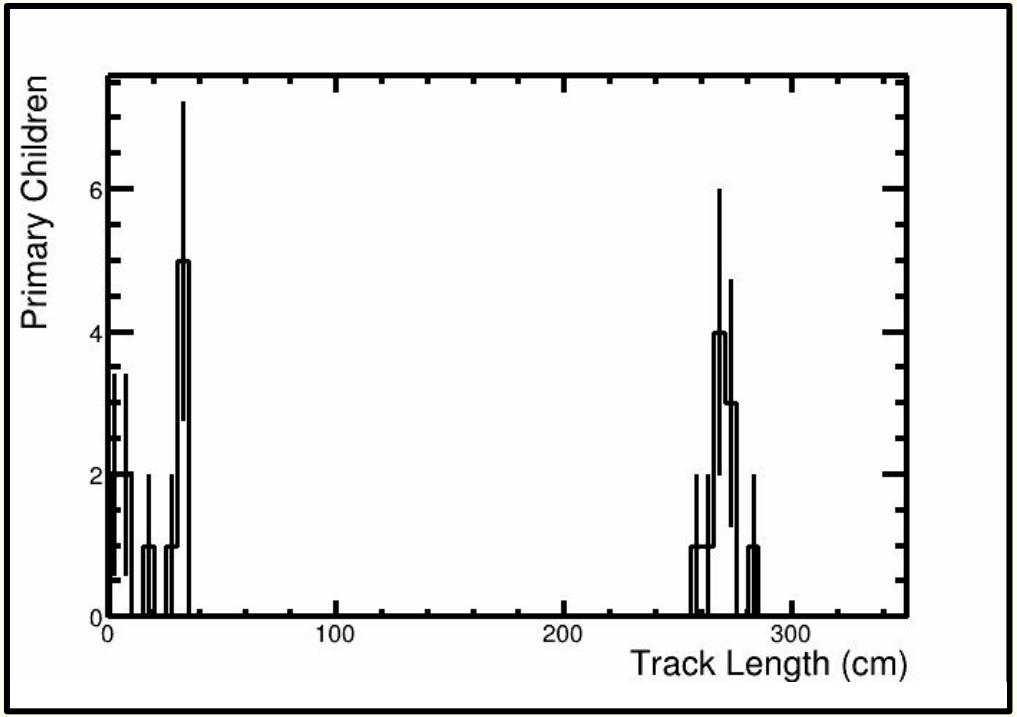
```
// Tell the loader we've declared all our spectra, time to fill them!
loader.Go();

// Child track length plot
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");
```

Let's save our plot in the directory we made earlier. I have chosen to save it as both a png and a pdf.

You did it!

You should now be the proud owner of one track length plot, look after it well!

If you've used the same file as you used in the analysis workshop it should also look identical!



Wrap Up

- Hopefully that wasn't too much of a whirlwind!
- Key takeaways:
  - 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.
  - If you are a python fan, Sungbin has written a wonderful framework for analysing SBND CAFs in python, cafpyana. See: https://github.com/sungbinoh/cafpyana/wiki

Wrap Up

- Hopefully that wasn't too much of a whirlwind!
- Key takeaways:
  - 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.
  - If you are a python fan, Sungbin has written a wonderful framework for analysing SBND CAFs in python, cafpyana. See: https://github.com/sungbinoh/cafpyana/wiki
- I have provided a "hidden" CAFAna macro in the CAF directory of the workshop area
  - ls -la $MRB_SOURCE/sbndcode/sbndcode/Workshop/CAF/
  - This macro and associated headers allows you to produce all the plots from the analyzer tutorial (and more).
  - In doing so it gives examples of 2D plots as well as using SpillCuts, SpillVars and truth information.
  - Backup slides have information on this…

Outline

Part 1: What are CAFs?

Part 2: [illegible] SBND CAFs have?

Part 4: Tutorial Extension

[illegible]utorial, how do I make and use the CAFs?

Side Note: Style

I added a little function to my macro to control a few global style parameters. Note, the best way to do this consistently across all your macros is to use the rootlogon.C file.

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

New includes

New function which is called at the start of the main function.

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

  gROOT->ForceStyle();
}

void Make_CAF_Plots()
{
  SetupStyle();
```

Plot 2: dE/dx vs. Residual Range

The next plot we wanted to make is the 2D distribution of dE/dx against residual range for all our tracks. In the macro we will need a new include, 2 new binning schemes, a new spectrum and a new plotting block. You can work out where to put them…

```
#include "TH2.h"
```

```
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);
```

```
// 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");
```

Plot 2: dE/dx vs. Residual Range

The next plot we wanted to make is the 2D distribution of dE/dx against residual range for all our tracks. In the macro we will need a new include, 2 new binning schemes, a new spectrum and a new plotting block. You can work out where to put them…

```
#include "TH2.h"
```

```
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);
```

```
// 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");
```

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

Plot 2: dE/dx vs. Residual Range

The next plot we wanted to make is the 2D distribution of dE/dx against residual range for all our tracks. We also need the two variables in our Var file!

```
const MultiVar kChildTrackdEdx([](const caf::SRSliceProxy *slc) -> std::vector<double> {
    std::vector<double> dEdx;

    for(auto const &pfp : slc->reco.pfp)
      {
        if(!pfp.parent_is_primary || pfp.parent == -1)
          continue;

        // Only interested in the collection plane (2)
        for(auto const& point : pfp.trk.calo[2].points)
          dEdx.push_back(point.dedx);
      }

    return dEdx;
  });

const MultiVar kChildTrackResRange([](const caf::SRSliceProxy *slc) -> std::vector<double> {
    std::vector<double> rr;

    for(auto const &pfp : slc->reco.pfp)
      {
        if(!pfp.parent_is_primary || pfp.parent == -1)
          continue;

        // Only interested in the collection plane (2)
        for(auto const& point : pfp.trk.calo[2].points)
          rr.push_back(point.rr);
      }

    return rr;
  });
```

Plot 2: dE/dx vs. Residual Range

The next plot we wanted to make is the 2D distribution of dE/dx against residual range for all our tracks. We also need the two variables in our Var file!

```
const MultiVar kChildTrackdEdx([](const caf::SRSliceProxy *slc) -> std::vector<double> {
    std::vector<double> dEdx;

    for(auto const &pfp : slc->reco.pfp)
      {
        if(!pfp.parent_is_primary || pfp.parent == -1)
          continue;

        // Only interested in the collection plane (2)
        for(auto const& point : pfp.trk.calo[2].points)
          dEdx.push_back(point.dedx);
      }

    return dEdx;
  });

const MultiVar kChildTrackResRange([](const caf::SRSliceProxy *slc) -> std::vector<double> {
    std::vector<double> rr;

    for(auto const &pfp : slc->reco.pfp)
      {
        if(!pfp.parent_is_primary || pfp.parent == -1)
          continue;

        // Only interested in the collection plane (2)
        for(auto const& point : pfp.trk.calo[2].points)
          rr.push_back(point.rr);
      }

    return rr;
  });
```

We use the same selection as the track length plot.

It is important we use the same selection in both of these Vars, otherwise we will have two different length vectors and the correspondence between values will be lost.

Plot 2: dE/dx vs. Residual Range

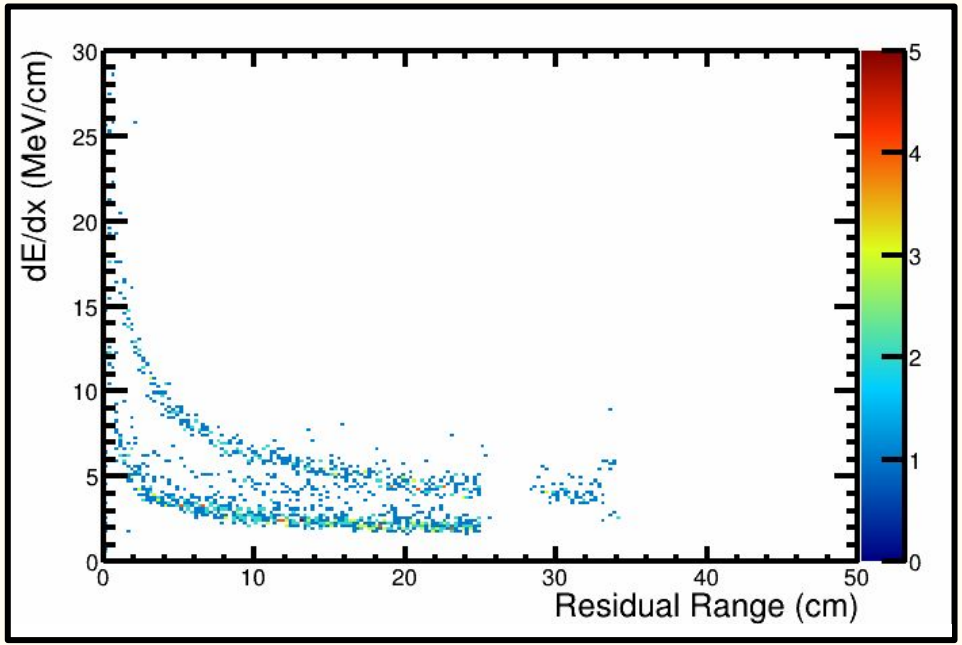
The next plot we wanted to make is the 2D distribution of dE/dx against residual range for all our tracks. We also need the two variables in our Var file!

```
const MultiVar kChildTrackdEdx([](const caf::SRSliceProxy *slc) -> std::vector<double> {
    std::vector<double> dEdx;

    for(auto const &pfp : slc->reco.pfp)
      {
        if(!pfp.parent_is_primary || pfp.parent == -1)
          continue;

        // Only interested in the collection plane (2)
        for(auto const& point : pfp.trk.calo[2].points)
          dEdx.push_back(point.dedx);
      }

    return dEdx;
  });

const MultiVar kChildTrackResRange([](const caf::SRSliceProxy *slc) -> std::vector<double> {
    std::vector<double> rr;

    for(auto const &pfp : slc->reco.pfp)
      {
        if(!pfp.parent_is_primary || pfp.parent == -1)
          continue;

        // Only interested in the collection plane (2)
        for(auto const& point : pfp.trk.calo[2].points)
          rr.push_back(point.rr);
      }

    return rr;
  });
```

We've chosen the collection plane, like we did in the analyzer tutorial.

We can then loop through the calorimetry points to get the dE/dx and residual range values.

Plot 2: dE/dx vs. Residual Range



This one *does* look a little different to the one we made earlier.

It still has the two populations, but it cuts off at 25cm and then has some smattering around 30cm from the higher ionising set.

This is because the CAFs only keep the last 25cm and first 5cm of calorimetry for each track (to save space) ( ).

A healthy reminder to investigate all features of your plots, and think about code-based reasons as well as physics reasons for what you're seeing.

Plots 3 & 4: Separating the longest track

We now want to separate our track length and dE/dx vs. residual range plots by which track is longer. A simple particle identification technique. This gives us a chance to call one Var within another Var.

```
const Var kLongestTrack([](const caf::SRSliceProxy *slc) -> int {
    double maxLength = std::numeric_limits<double>::lowest();
    int maxLengthID  = std::numeric_limits<int>::signaling_NaN();

    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 we make a Var that returns the index of the longest track in that slice.

Plots 3 & 4: Separating the longest track

We now want to separate our track length and dE/dx vs. residual range plots by which track is longer. A simple particle identification technique. This gives us a chance to call one Var within another Var.

```
const Var kChildTrackLengthLongestTrack([](const caf::SRSliceProxy *slc) -> double {
    int maxLengthID = kLongestTrack(slc);

    if(maxLengthID == std::numeric_limits<int>::signaling_NaN())
      return std::numeric_limits<double>::signaling_NaN();

    return slc->reco.pfp[maxLengthID].trk.len;
  });

const MultiVar kChildTrackLengthOtherTracks([](const caf::SRSliceProxy *slc) -> std::vector<double> {
    std::vector<double> trackLengths;

    int maxLengthID = 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 track length of the longest track and a MultiVar that returns the track lengths of the other tracks.

Plots 3 & 4: Separating the longest track

We now want to separate our track length and dE/dx vs. residual range plots by which track is longer. A simple particle identification technique. This gives us a chance to call one Var within another Var.

```
Spectrum *sChildTrackLengthLongestTrack = new Spectrum("sChildTrackLengthLongestTrack", trackLengthBins, loader,
                                                       kChildTrackLengthLongestTrack, kNoSpillCut, kIsNeutrinoCandidateSlice);
Spectrum *sChildTrackLengthOtherTracks = new Spectrum("sChildTrackLengthOtherTracks", trackLengthBins, loader,
                                                      kChildTrackLengthOtherTracks, kNoSpillCut, kIsNeutrinoCandidateSlice);
Spectrum *sChildTrackdEdxResRangeLongestTrack = new Spectrum("sChildTrackdEdxResRangeLongestTrack", loader, resRangeBins,
                                                             kChildTrackResRangeLongestTrack, dEdxBins, kChildTrackdEdxLongestTrack,
                                                             kNoSpillCut, kIsNeutrinoCandidateSlice);
Spectrum *sChildTrackdEdxResRangeOtherTracks = new Spectrum("sChildTrackdEdxResRangeOtherTracks", loader, resRangeBins,
                                                            kChildTrackResRangeOtherTracks, dEdxBins, kChildTrackdEdxOtherTracks,
                                                            kNoSpillCut, kIsNeutrinoCandidateSlice);
```

We will need two spectrums for each of these plots… One for the longest track and one for the other tracks.

Plots 3 & 4: Separating the longest track

We now want to separate our track length and dE/dx vs. residual range plots by which track is longer. A simple particle identification technique. This gives us a chance to call one Var within another Var.

```
Spectrum *sChildTrackLengthLongestTrack = new Spectrum("sChildTrackLengthLongestTrack", trackLengthBins, loader,
                                                       kChildTrackLengthLongestTrack, kNoSpillCut, kIsNeutrinoCandidateSlice);
Spectrum *sChildTrackLengthOtherTracks = new Spectrum("sChildTrackLengthOtherTracks", trackLengthBins, loader,
                                                      kChildTrackLengthOtherTracks, kNoSpillCut, kIsNeutrinoCandidateSlice);
Spectrum *sChildTrackdEdxResRangeLongestTrack = new Spectrum("sChildTrackdEdxResRangeLongestTrack", loader, resRangeBins,
                                                             kChildTrackResRangeLongestTrack, dEdxBins, kChildTrackdEdxLongestTrack,
                                                             kNoSpillCut, kIsNeutrinoCandidateSlice);
Spectrum *sChildTrackdEdxResRangeOtherTracks = new Spectrum("sChildTrackdEdxResRangeOtherTracks", loader, resRangeBins,
                                                            kChildTrackResRangeOtherTracks, dEdxBins, kChildTrackdEdxOtherTracks,
                                                            kNoSpillCut, kIsNeutrinoCandidateSlice);
```

Back in the macro, we will need two spectrums for each of these plots… One for the longest track and one for the other tracks.
We're also going to need this include in a moment, so we can make a legend to distinguish the two components.

```
#include "TLegend.h"
```

Plots 3 & 4: Separating the longest track

We now want to separate our track length and dE/dx vs. residual range plots by which track is longer. A simple particle identification technique. This gives us a chance to call one Var within another Var.

```
// Separated track length plot
TCanvas *cChildTrackLengthSeparated = new TCanvas("cChildTrackLengthSeparated", "cChildTrackLengthSeparated");
cChildTrackLengthSeparated->cd();

TH1D *hChildTrackLengthLongestTrack = sChildTrackLengthLongestTrack->ToTH1(sChildTrackLengthLongestTrack->Livetime(), kLivetime);
hChildTrackLengthLongestTrack->SetTitle(";Track Length (cm);Primary Children");
hChildTrackLengthLongestTrack->SetLineColor(kMagenta+2);
hChildTrackLengthLongestTrack->Draw("histe");

TH1D *hChildTrackLengthOtherTracks = sChildTrackLengthOtherTracks->ToTH1(sChildTrackLengthOtherTracks->Livetime(), kLivetime);
hChildTrackLengthOtherTracks->SetLineColor(kOrange+2);
hChildTrackLengthOtherTracks->Draw("histesame");

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

cChildTrackLengthSeparated->SaveAs(saveDir + "/child_track_length_separated.pdf");
cChildTrackLengthSeparated->SaveAs(saveDir + "/child_track_length_separated.png");
```

We have a longer plotting block this time! This is the track length

Plots 3 & 4: Separating the longest track

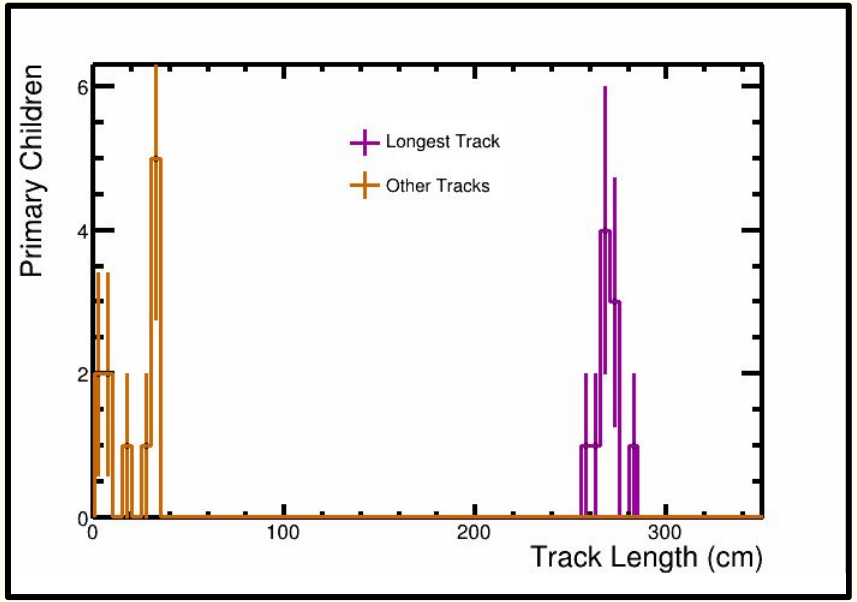
We now want to separate our track length and dE/dx vs. residual range plots by which track is longer. A simple particle identification technique. This gives us a chance to call one Var within another Var.

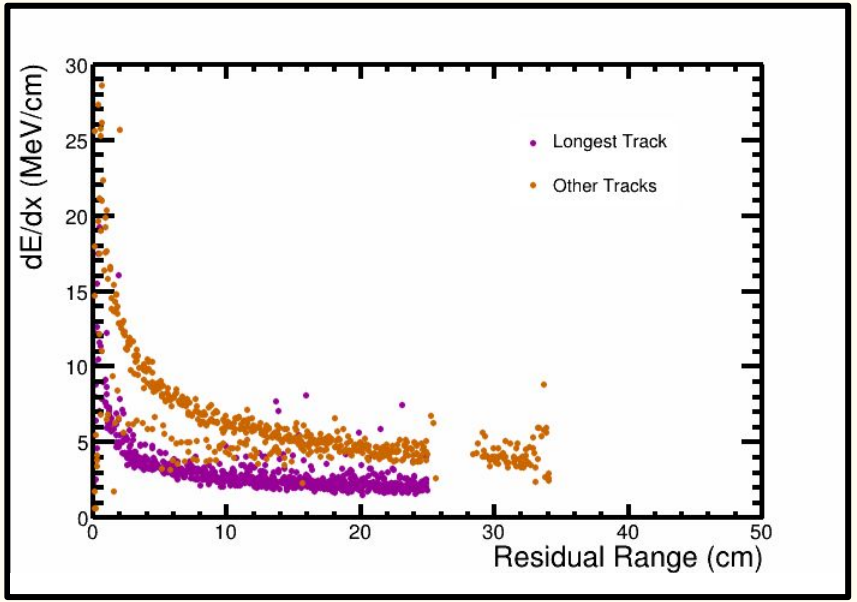
```
// Separated track length plot
TCanvas *cChildTrackLengthSeparated = new TCanvas("cChildTrackLengthSeparated", "cChildTrackLengthSeparated");
cChildTrackLengthSeparated->cd();

TH1D *hChildTrackLengthLongestTrack = sChildTrackLengthLongestTrack->ToTH1(sChildTrackLengthLongestTrack->Livetime(), kLivetime);
hChildTrackLengthLongestTrack->SetTitle(";Track Length (cm);Primary Children");
hChildTrackLengthLongestTrack->SetLineColor(kMagenta+2);
hChildTrackLengthLongestTrack->Draw("hist");

TH1D *hChildTrackLengthOtherTracks = sChildTrackLengthOtherTracks->ToTH1(sChildTrackLengthOtherTracks->Livetime(), kLivetime);
hChildTrackLengthOtherTracks->SetLineColor(kOrange+2);
hChildTrackLengthOtherTracks->Draw("histesame");

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

cChildTrackLengthSeparated->SaveAs(saveDir + "/child_track_length_separated.pdf");
cChildTrackLengthSeparated->SaveAs(saveDir + "/child_track_length_separated.png");
```

We set different line colo**u**rs and make a legend to help distinguish the two. “le” means include the line and error line in the legend display.

Plots 3 & 4: Separating the longest track

This went through the track length plot, the dE/dx vs. residual range plot is very similar. You can work it out! Remember the solutions are available in the workshop branch!





Plot 5: OpT0 Finder Time

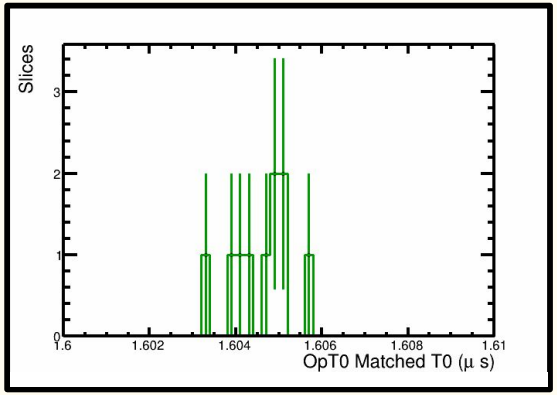
This one actually becomes very straightforward!

```
const Var kOpT0Time([](const caf::SRSliceProxy *slc) -> double {
    return slc->opt0.time;
  });
```

```
Binning opt0TimeBins    = Binning::Simple(50, 1.6, 1.61);
```

```
Spectrum *sOpT0Time = new Spectrum("sOpT0Time", opt0TimeBins, loader, kOpT0Time, kNoSpillCut, kIsNeutrinoCandidateSlice);
```

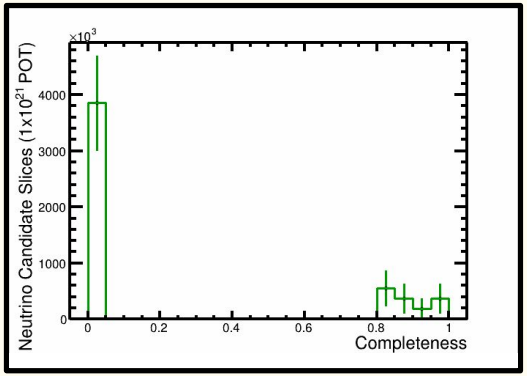
```
// Child track length plot
TCanvas *cOpT0Time = new TCanvas("cOpT0Time", "cOpT0Time");
cOpT0Time->cd();

TH1D *hOpT0Time = sOpT0Time->ToTH1(sOpT0Time->Livetime(), kLivetime);
hOpT0Time->SetTitle(";OpT0 Matched T0 (#mu s);Slices");
hOpT0Time->SetLineColor(kGreen+2);
hOpT0Time->Draw("histe");

cOpT0Time->SaveAs(saveDir + "/opt0_time.pdf");
cOpT0Time->SaveAs(saveDir + "/opt0_time.png");
```

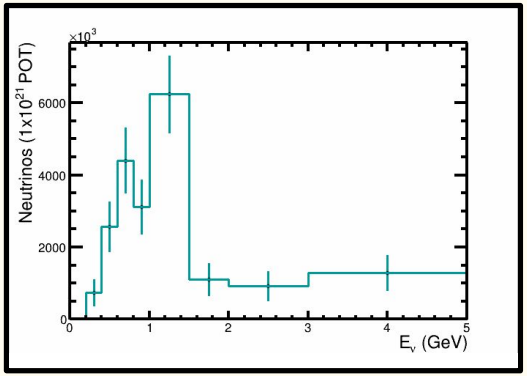


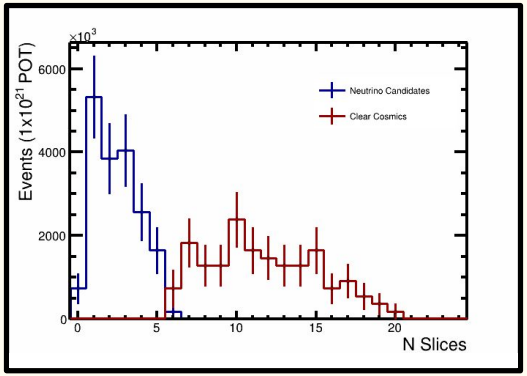
Extra Plots

In the sbndcode/Workshop/CAF/ directory you should find the solutions. You will also find a second set of files Extra_Plots.C ExtraVars.h ExtraCuts.h

In these files you will find examples that use SpillVar and SpillCut it also introduces POT scaling, truth information, neutrino event files and variable binning.



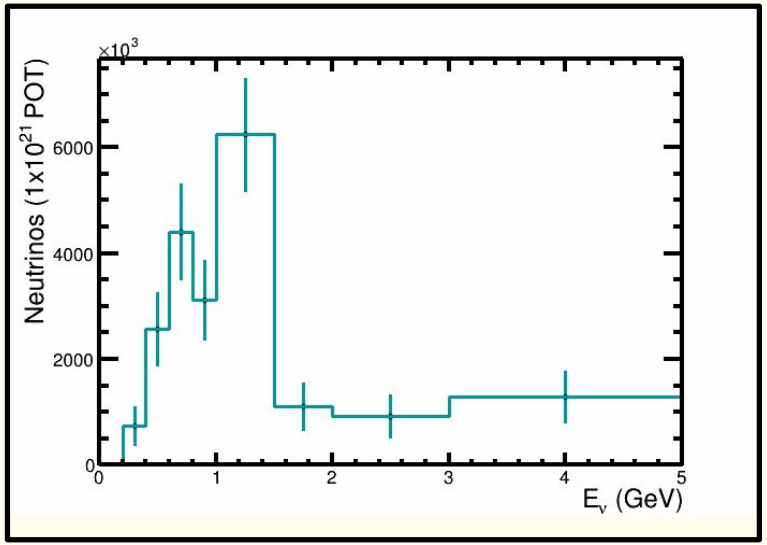




Extra Plot 1

Plot of true neutrino energy.

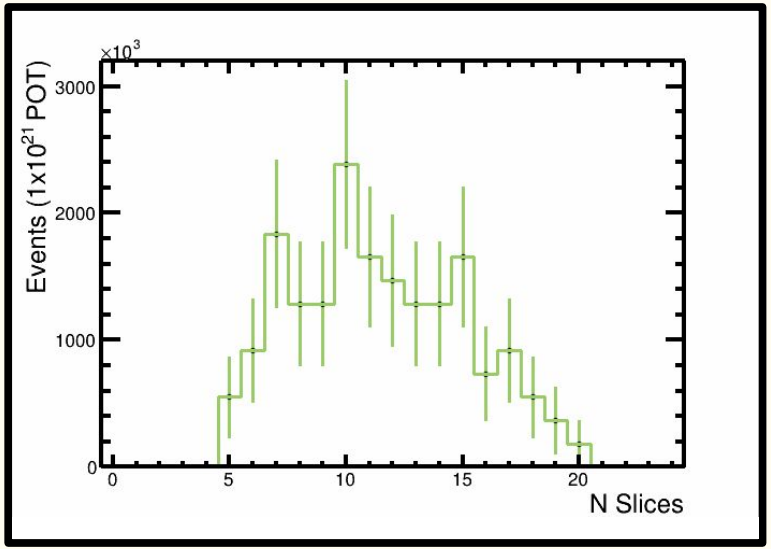
- Note we are reading a different CAF file. Not the 1μ1p files we made but a GENIE neutrino file I made for you.
- For the first time we've used `Binning::Custom` instead of `Binning::Simple` to provide these variable length bins.
- I have 'POT-scaled these plots' (check the `ToTH1()` functions). I chose 1x10^{21} POT, the total expected exposure of SBND.
- The `SpillMultiVar` used returns all true neutrino energies from this event.



Extra Plot 2

Plot of total number of slices

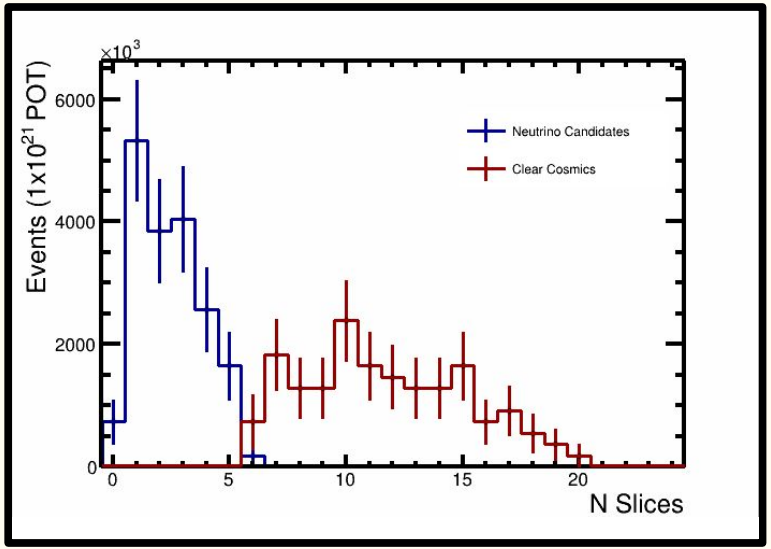
- This looks at the total number of slices in this event.
- Note that, despite the POT scaling, the statistical error bars have been preserved to the sample scale.



Extra Plot 3

Plot of number of neutrino candidate compared to clear cosmic slices.

- Splitting the total number of slices into Pandora's "neutrino candidate" and "clear cosmic" categories.
- Use of a TLegend again.
- The clear cosmics histogram was made by an example of subtracting one Spectrum from another, this saved making an extra Spectrum to be filled during running!



Extra Plot 4

Plot of some neutrino candidate slices' completenesses

- Completeness is a measure of how much of the true neutrino activity made it into that slice.
- To demonstrate the use of both a SpillCut and Cut in the same Spectrum I made a very unrealistic selection (only neutrino candidate slices are plotted for events with exactly one neutrino candidate slice). Normally you would make a much more sophisticated choice!