Data Structures in ROOT

ROOT @ TUNL (R @ T)

Sean Stave
Overview

• ROOT Trees
  – ntuples, creating trees and reading trees
• 1d-Cuts and Graphical Cuts
  – Create, Save, and Use
• Using arrays in ROOT
• Histogram Operations
  – Cuts, Fitting, Integral, Contents
• TPhysics
  – Lorentz Transforms, etc
• TMath
  – Special Functions
ROOT Trees

- Trees are collections of objects typically arranged event by event
- Called ntuples in PAW
- Like a spreadsheet where each row is a different event and each column is a piece of data

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Type</th>
<th>ADC1</th>
<th>ADC2</th>
<th>TDC1</th>
<th>TDC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3E</td>
<td>-DD</td>
<td>3E</td>
<td>99</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3E</td>
<td></td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td>3E</td>
<td>97</td>
<td>967</td>
</tr>
</tbody>
</table>
• Quick, simple tree with up to 12 branches:
  – TNtuple *nt = new TNtuple("nt","Title","a:b:c");

• Use by:
  – nt->Draw("a");
  – nt->Draw("b:a","a<5");
  – nt->Draw("b:a","(a+b)<5","colz");

• See myNtuple.C
• When using the Draw command, ROOT will automatically decide on the limits of the plot
• Sometimes, not what you want (too wide, too coarse)
• There are two ways to define it yourself:
  – Create a histogram called h with 3000 bins from 1 to 10:
    ```
    nt->Draw("a>>h(3000,1,10)")
    ```
  – Do the same thing:
    ```
    TH1F *h = new TH1F("h","h",3000,1,10);
    nt->Draw("a>>h");
    ```
• Both commands redirect the Draw output to a histogram of your choosing
**TTrees**

- Most experiments will need more than 12 branches so must use a different object

- Declare a tree:
  - `TTree *tree = new TTree("tree","Title");`

- Add branches to the tree:
  - `Float_t x,y,z;`  
    - Variables that will be used to fill tree
  - `tree->Branch("x",&x,"x/F");`

- Unlimited amount of branches can be added
- Other variable types can be added to the tree
• Set the variables to the values you want:
  – x=4; y=2; z=8;
• Then fill the tree
  – nt->Fill();
  – It will automatically go through all of the declared branches and fill their values
• Can save the tree to a ROOT file in the same way we save a histogram or a canvas
  TFile *f = new
      TFile("myNtuple.root","RECREATE");
  tree->Write();
  f->Close();
• See myTree.C
Looping over TTrees

- Already showed how to draw a tree and generate some plots but slow and cumbersome for many histograms
- Can access branches event by event and fill histograms
- Nice method for getting trees from a file is using TChain
  ```c++
  TChain *nt = new TChain("tree");
  ```
  Name of the tree in the file

Then just add the ROOT files which have the trees you want:
```c++
nt->Add("myNtuple.root");
```
• Now you can use the tree just like normal:
  – nt->Draw("x:y");

• Or you can access each branch event by event

• Similar way to declaring the tree

• Declare the ntuple variables (can be named anything but I typically use a prefix nt_ so that I don't get them confused with other variables):
  Float_t nt_x, nt_y, nt_z;

• Then set the branch addresses:
  nt->SetBranchAddress("x",&nt_x);

Can now use nt->GetEntry(i) where i is the event number to set the ntuple variables equal to the branch values for that event (Use nt->GetEntries() to find out how many events are in the file)

• See myTree2.C
Cuts in ROOT

- Have already seen simple command line cuts:
  - `nt->Draw("x","y>5");`
  - Can be complicated and long statements
- Can still use these but, again, is typically slow for larger data sets
- More efficient method is to use logical statements while filling the histograms in a loop
After you have the tree loaded and the branches set, can write a simple loop:

```c
Int_t i, nentries, nbytes;
nentries=(Int_t)nt->GetEntries();
for (i=0;i<nentries;i++){
    nbytes=nt->GetEntry(i);
    (Variables now have the values for this event and you can fill histograms)
}
```

See myTree3.C and myTree4.C
Setting Graphical Cuts in ROOT

- Can define higher dimensional cuts in ROOT using algebra and logic
- Can also define 2D graphical cuts and apply them
- This is the TCutG class:
  ```
  TCutG *cut = new TCutG();
  cut = (TCutG*) gPad->WaitPrimitive("CUTG")
  ```
- Click on the scissors on the ROOT Toolbar to select the graphical cut tool and then select your region
- The WaitPrimitive command returns a TObject and we must cast it into a TCutG so that nothing complains about being the wrong data type
- See `set_myCutG.C`
Using Graphical Cuts in ROOT

- Once the TCutG is defined, it can be saved to a ROOT file just like any other object.

- To use it, you want to load the file and then the cut:
  
  ```
  TCutG *cut;  // (pointer to cut)
  TFile *f2 = new TFile("myCutG.root");  // (ROOT file)
  cut = (TCutG*)f2->Get("cut");  // (Grab cut from file)
  ```

- The TFile class has a Get method which allows you to grab items from it. They also must be cast into the right data type or else ROOT will get confused.

- The cut can then be used with the simple Draw command:
  
  ```
  nt->Draw("y:x","cut");
  ```
• Or you can use 
  \texttt{cut->IsInside(var1, var2)}
  in the event loop to select the events that are inside (or outside) the cut
• Must be careful to use \texttt{var1} and \texttt{var2} in the same order that you used to define the cut
• See \texttt{myCutG.C}
Using Arrays in ROOT

Arrays are used in C++ to conveniently access indexed variables

- Eg.: For variables called, h1, h2, h3, inconvenient to call them in a loop
- If you call them h[1], h[2], h[3] then you can place a variable in the brackets and access them

To use:

- Declare an array of pointers and then define the histograms
  - TH1F *h[10];
  - for (Int_t i=0;i<10;i++){
    h[i] = new TH1F(Form("h[%i]",i),
                   Form("Title%i",i),
                   100,0,10);
  }
  - Can then access them with variables such as h[i]
  - See myArray.C
  - Can apply to any ROOT objects (TFiles, TGraphs, TCutG, etc)
Histogram Operations

- Have already seen the Draw() command
- Has many options:
  - col : Colored squares
  - colz : colored squares with a z axis scale
  - hist : histogram
  - surf : surface plot
  - cont : contour plot
  - same : plot on the existing histogram (for overlaying multiple histograms)
Histogram Operations

- Sometimes, want to access the properties or values of a histogram.
- Often the method is self-explanatory you just need to know about it and its syntax.
- Some methods are:
  - `GetBinContents(binx, biny)`
  - `GetBinCenter(binx, biny)`
  - `GetBinError(binx, biny)`
  - `GetNbinsX()`
  - `GetNbinsY()`
  - `Scale(factor)`
  - `Integral(start_bin, end_bin)`

- 1-D histograms only have 1 argument for the bin. 2- and 3-D histograms need more arguments.
Histogram Operations

- We will take Mohammad's myHist-2.C and look at the histograms in detail
- See my-Hist4.C for examples
- The first bin in a histogram is number 1. The last bin of a 1-D histogram can be found using h->GetNbinsX()
- The 0\textsuperscript{th} and last+1 bins are the underflow and overflow bins, respectively
- Can Add or Scale histograms
  - h->Scale(2.0); (will scale each bin in h by 2.0)
  - h->Add(h2); (Will add h2 to h)
- Can Rebin histograms
  - h->Rebin(factor); (will rebin by that factor)
Histogram Operations

- Integrals have two steps in ROOT
- First, must identify the bins to integrate over
- Then, can integrate using h->Integral()
- This is to remind you that the histogram integral is only an approximation due to the finite bin width in the histogram

Example in my-Hist4.C

- Integral of Hist1 from 1.0 to 3.0
- Int_t low_bin=Hist1->FindBin(1.0);
- Int_t high_bin=Hist1->FindBin(3.0);
- Double_t sum=Hist1->Integral(low_bin,high_bin);

In this case, integral is really between the low edge of bin 56 and the high edge of bin 66
Fitting Histograms

- As usual, there are quick, simple, less flexible ways and more complicated ways which are more flexible but more prone to errors.
- Quick way to fit a histogram is with the Fit() command: `Hist1->Fit("gaus")`
- `gaus` is a predefined function.
- Also have:
  - `polN` : polynomial of order N, up to 9
  - `expo` : exponential
  - User defined functions
- See `myFit-1.C`
User Defined functions

- Declare your own TF1 object
  - TF1 *fit1 = new TF1("fit1",
    "[0]*exp(-0.5*(([1]-x)/[2])^2)",-5,5);
  - Parameters are in brackets
  - Last arguments are the range, low to high
- Might need to set initial parameters (fitter needs a good starting guess)
  - fit1->SetParameters(...,...,etc)
- Then run:
  - Hist1->Fit("func") where func is the name of the TF1 object you made
- See myFit-2.C
Fitting with MINUIT

- Sometimes will need the full power of MINUIT
- Originally developed in FORTRAN and a part of CERNLIB
- Has been ported to ROOT but still accessed in a similar way to the older version
- Able to minimize much more complicated functions
Chi$^2$ Minimization Theory

- The user provides a function to MINUIT which returns a chi$^2$

$$
\chi^2 = \sum_i \left( \frac{f(x_i, \text{par}[i]) - y_i}{\sigma_i} \right)^2
$$

- Chi$^2$ just a way of determining how close your fitting function is to the data

- For proper fitting function and statistically distributed events, expect approximately a Chi$^2$ of 1 for each data point (histogram bin)
Chi$^2$ Minimization Theory

- The fitting function has several parameters which MINUIT will vary and see how that affects the output
- MINUIT is reasonably smart and will seek out the minimum chi$^2$ by varying the parameters in such a way that minimizes the number of calls
- The chi$^2$ distribution is like a bowl and MINUIT will hopefully find the lowest point in that bowl
- Good starting values are essential or else it can get caught in a local minimum
Fitting with MINUIT

- Must load the MINUIT library:
  - gSystem->Load("libMinuit.so");
- Must define your own chi2 function (typically called fcn)
- Declare a TMinuit object:
  - TMinuit *gMinuit= new TMinuit(2);
- Tell MINUIT which minimization function to use:
  - gMinuit->SetFCN( fcn );
- Initialize the fitting parameters and the fitter (see myFitMINUIT.C)
- Set the \( \Delta \chi^2 \) correctly (more advanced concept)
  - 1 par: \( \Delta \chi^2 =1 \), 2 par: \( \Delta \chi^2 =2.4 \), 3 par: \( \Delta \chi^2 =3.6 \)
- Execute the minimization
  - gMinuit->mnexcm("MIGRAD", arglist, 2, ierflg );
- Get the parameter values
- See myFitMINUIT.C for details
Fitting with MINUIT

• Remember: the fcn can be as simple or as complex as you want it to be
• Depending on the complexity, MINUIT may converge rapidly and slowly
Fitting with MINUIT

Word of warning:

- Fitting properly really takes practice to get it right
- Think of other statistical tests to check your fit
- Example:
  - Phil generates pseudodata for 1000's of experiments and fits each one
  - He then looks at the distribution of the fit results and makes sure that they agree with the MINUIT estimates based upon a single experiment
- We are also considering some brute force scanning of the parameter space for certain applications
- Computers are fast so use them in whichever way gets you to your solution

Image from Genesis Mission/NASA
TPhysics

- Like TMinuit, requires a library to be loaded:
  - `gSystem->Load("libPhysics.so");`

- Contains classes like:
  - `TVector2` : 2D vectors
  - `TVector3` : 3D vectors
  - `TLorentzVector` : 4D vectors
  - `TRotation` : Allows rotations of vectors

- To use `TLorentzVector`:
  - Declare variable: `TLorentzVector v;`
  - Set the total energy and momentum:
    - `v.SetPxPyPz(Px,Py,Pz,E)`
  - Remember, all relativistic so use: \( E^2 = P^2 + M^2 \) (c=1 units)
  - Can then add the 4-vectors, find their magnitudes etc.
TPhysics

- See examples `myPhysics-1.C` (HIGS photon calculation) and `myPhysics-2.C` (11B(p,a) calculation)
- Note that you can return the energy, momentum, mass, angle, etc using the class:
  - `v.E()`  
  - `v.P()`  
  - `v.M()`  
  - `v.Theta()`  
  - `v.Beta()`  
- Angle units are radians
- Energy and momentum units are typically MeV but depends on how you define them
- Remember, the total magnitude of the 4-vector is preserved
TMath

- ROOT has many built-in mathematical functions
- Use as: TMath::Abs(x), TMath::CosH(x)
- May need additional functions found at: root.cern.ch/drupal/content/mathmore-library
- Needs to be loaded:
  gSystem->Load("libMathMore.so");
- See myTMath.C for more details
Homework

- A real data file has been created for you using a CODA to ROOT converter (future talk):
  - homework3.root

- For those who are curious, it is the output of neutron detectors looking at the $d(\gamma,n)p$ reaction with 15 MeV, circularly polarized photons

- It contains a tree which has many branches
  - 1: Load the ntuple file and take a look at it with the TBrowser
  - 2: Plot branch ADC1 vs. TAC1 (Remember: Draw("y:x"))
  - 3: Create a 2D graphical cut on the righthand band
  - 4: Save the cut to a file
  - 5: Apply the cut and check the ADC1 vs TAC1 plot
  - 6: Plot the ADC1 distribution with and without the cut