Histograms, Ntuples, Suez, EventStore, Batch Queues and Root!

Nadia Adam
Cornell University
What ARE Histograms and Ntuples?

- In high energy physics we deal with a lot of data!
- But how do we analyze it, or “see” what’s going on?

A histogram is just like an array. The variable to be plotted is divided into “bins” of the desired width. The histogram stores the number of events in each bin. And that’s basically it . . . the histogram allows us to see how our data is distributed!
What ARE Histograms and Ntuples?

• But in many cases it isn’t convenient, or even useful, to store just single variables in histograms.
• We generally want to look at many different variables, or see how a particular variable changes when “cuts” are placed on other variables.
• We need to store a whole lot of information in parallel - and this is what an ntuple (“en-toople” or “en-tuple” or “tree”!) does for us.

```
Event

1
v1
v2
v3
v4

2
v1
v2
v3
v4

3
v1
v2
v3
v4

4
v1
v2
v3
v4

5
v1
v2
v3
v4

6
v1
v2
v3
v4

...```
More about Ntuples ...

- From an ntuple we can make histograms or 2D “scatter” plots!
- **Example:**
  - In our ntuple we store $v1$ and $v2$.
  - I can plot $v1$ straight or I can look at $v1$ only for those events where $x < v2 < y$ (or any other more complicated constraint you might like to think of!).

<table>
<thead>
<tr>
<th>$D^0 \rightarrow K^- \pi^+$</th>
<th>$D^0 \rightarrow K^- \pi^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="histogram.png" alt="" /></td>
<td><img src="scatter_plot.png" alt="" /></td>
</tr>
</tbody>
</table>

No Cut! $|\Delta E| < 0.01$
But How Do I Get the Data In There?

- CESR + CLEO = Raw Data
- Raw Data (on tape)
- Calibration (alignment, t0’s, drift functions, etc ...)
- PASS2 (Find tracks, showers, etc)
- Analysis quality (physics grade) data in EventStore
- Your C++ analysis code (in SUEZ)
- Store skim data in pds AND ROOT files!
- Use **ROOT** to analyze data further: make plots, fit, etc ...

**What we’ll talk about today!**
A Little History ...

• Before we start on the exact details of how to make ROOT histograms and ntuples in SUEZ code there are a couple of things to note!
• At CLEO most people are relatively new to using ROOT, everyone used to use PAW or MN_FIT.
  • So in SUEZ everything was set up to make PAW and MN_FIT style histograms (non-C++, non-Object Oriented!).
• Now there are some very nice ways to convert these old style histograms and ntuples into the ROOT equivalents ....
  • **OR we can put things directly into ROOT (slightly more powerful for some things I believe).**
• I will try (bravely) to show you both ways, but will focus on using ROOT directly, especially for making Trees, where ROOT is much more powerful than the old programs since it can store Objects as branches!
Simple Histograms in SUEZ

The easiest way to make a simple histogram in SUEZ is probably not to use ROOT directly, but to use the SUEZ HistogramInterface classes:

1. **Declare a histogram as member data in your Processors *.h file**

   ```
   #include "HistogramInterface/HistogramPackage.h"
   ...
   // Suez style histograms!
   HIHist1D *m_kpiMass;
   ```

2. **Define histogram in the hist_book method of the *Proc.cc file**

   ```
   void RootExampleProc::hist_book( HIHistoManager& iH )
   {
       // book your histograms here
       m_kpiMass = iH.histogram( "D^0 -> K pi M_{bc} (GeV)", 20, 1.83, 1.89 );
   }
   ```
3. Fill the histogram in the event method of *Proc.cc file

```
// Fill our example suez histogram
if ( m_tag->decayMode() == 0 )
{
    m_kpiMass->fill( m_tag->beamConstrainedMass() );
}
```

4. Load the conversion module in your tcl script

```
module sel RootHistogramModule
root file root_suez_style_example_$env(DATABASE).root
root init
```
Simple Ntuples in SUEZ

To make a simple Ntuple (which can store only float/double type variables) we can again use the SUEZ HistogramInterface classes:

1. Declare an ntuple as member data in your Processors *.h file

```
#include "HistogramInterface/HistogramPackage.h"
...
// Suez style histograms!
HIHist1D *m_shwrNtuple;
```

2. Define ntuple variable list in as an enum in your *.h file

```
enum {
    ksp,
    ksenergy,
    ksmatch,
    kNumVar
};
```
Simple Ntuples in SUEZ

3. Define the ntuple in the hist_book method of *Proc.cc file

```
HINtupleVarNames names(kNumVar);
names.addVar( ksp, "sp" );
names.addVar( ksenergy, "senergy" );
names.addVar( ksmatch, "smatch" );

m_shwrNtuple = iH.ntuple( 99, "Showers", kNumVar, 10000, names.names() );
```

4. Fill the ntuple in the event method of *Proc.cc file

```
// Suez style - array for filling
float suez_tuple[ kNumVar ];

for(NavShowerConstIt shwr = m_showers.begin(); shwr != m_showers.end(); shwr++)
{
    ...
    suez_tuple[ ksp ] = photon.pmag();
    suez_tuple[ ksenergy ] = photon.energy();
    suez_tuple[ ksmatch ] = (float)(!shwr->noTrackMatch());
    m_shwrNtuple->fill( suez_tuple );
}
```
Before we advance to the slightly more complicated task of how to fill ROOT objects directly in SUEZ we need a little background!

ROOT is C++ based, so everything in ROOT is an object or class and there is a lot of inheritance going on ... but let’s try and see what we’re most interested in!

- In ROOT the basic histogram object is a class called TH1
- But! You will almost never use TH1 directly, instead you use higher level classes which specify the data type:
  - TH1F (float type)
  - TH1D (double type) etc
- The class constructor takes a name, title, num bins, low bin, high bin
- Contains many functions, e.g. Draw()

As we’ve already hinted in ROOT the ntuple like object is called a “tree”
- The reference is obvious since a “tree” has many “branches” (in this case variables)
- The basic class is TTree
- The variables stored in the tree are held in branch objects ... the TBranch class
- TTree object again contains many functions, e.g., Draw(“var_name”)
Filling ROOT Objects Directly in SUEZ

```
// In *.h file
#include "TFile.h"
#include "TH1F.h"
...
TH1F *m_hist;
TFile *m_file;

// In *.cc file
/home/nea9/Source/RootExampleProc/Makefile

// In init() method
m_file = new TFile("/home/me/mydir/myfile.root","RECREATE");
m_hist = new TH1F("h1","My Title",nbins,low,high);

// In event() method
m_hist->Fill( myVar );

// In terminate() method
m_file->Write();
m_file->Close();
```

The Basics!

You will need to make sure you have all the ROOT libraries linked correctly in your Makefile!
Filling a TTree from SUEZ

Make a class to hold all the “variables” to go in the tree (remember these can be more than just simple floats now, can be anything including classes!)

MyNtuple

Make a class whose job is to declare the tree branches, fill the tree using the variable holder class (MyNtuple) and write the finished tree to a file.

MyFillTree

Use the Processor to access data and fill the tree variables, using the fill class (MyFillTree).

We will follow this algorithm, which packages things fairly neatly. But of course ... there is always more than one way to skin a cat!
Filling a TTree from SUEZ

• Make the class to hold our variables ... *ExampleNtuple*

```cpp
class ExampleNtuple {
public:
    // -- constants, enums and typedefs --
    enum { MAXNSHWR = 100 };
    enum { MAXNTRK = 100 };

    // -- Constructors and destructor --
    ExampleNtuple();
    virtual ~ExampleNtuple();

    int run;
    int event;
    float beamE;

    // Track data
    int ntrk;
    float tp[MAXNTRK];
    float tpx[MAXNTRK];
    float tpy[MAXNTRK];
    float tpz[MAXNTRK];
    float tenergy[MAXNTRK];
    float tchi2[MAXNTRK];
    float tz0[MAXNTRK];
    float td0[MAXNTRK];
    ...
};
```

```cpp
ExampleNtuple::ExampleNtuple() {
    init();
}

ExampleNtuple::~ExampleNtuple() {
}

void ExampleNtuple::init() {
    // Initialize all Ntuple elements
    run = -999;
    ...
    // Track data
    ntrk = -999;
    for (int i = 0; i < MAXNTRK; i++) {
        tp[i] = -999;
        tpx[i] = -999;
        ...
    }
    ...
}
```

Notice, we have arrays!
Filling a TTree from SUEZ

- Make a class to fill the tree ... \textit{FillExampleTree}

```cpp
void FillExampleTree::init(std::string filename, ExampleNtuple* ntuplePtr)
{
    m_ntuple = ntuplePtr;
    m_file = new TFile( filename.c_str(), "RECREATE" );
    m_tree = new TTree( "rtexnt", "Root Example Ntuple", 1 );

    // Declare Branches!
    m_tree->Branch("run", &(m_ntuple->run), "run/I");
    ...
    m_tree->Branch("ntrk", &(m_ntuple->ntrk), "ntrk/I");
    m_tree->Branch("tpx", &(m_ntuple->tpx), "tpx[ntrk]/F");
    ...
}

void FillExampleTree::fill()
{
    m_tree->Fill();
    dout << "Tree entries " << m_tree->GetEntries() << endl;
}

void FillExampleTree::finalize()
{
    iout << "Here in finalize" << endl;
    m_file->Write();
    m_file->Close();
    iout << "Done write and close" << endl;
}
```

- Make the tree and the output file
- Fill ... called from event()
- Declare the Branches!
- We're done, write and close the file
#include "RootExampleProc/FillExampleTree.h"
...
void RootExampleProc::init( void )
{
    // Initialize the class which
    // fills the Root tree!!
    m_fillExample.init( m_fileName,
                        &m_exampleNtuple );
}

void RootExampleProc::terminate( void )
{
    // Finalize the Root Ntuple
    m_fillExample.finalize();
}

// Ntuple parameters
ExampleNtuple  m_exampleNtuple;
FillExampleTree m_fillExample;

set variables in holder class!

Set
variables
in holder class!

Finish up!

Done!

root tree!

Fill!

in

Take a look at the root tree!

Done!
Running Jobs - EventStore and Batch Queues!

- The analysis quality data is kept in the EventStore
- It is accessed at run time from the tcl script:

```
# Use the event store module sel EventStoreModule
eventstore in 20050604 dtag dataset data31
```
- There are many ways to access different types of data:
  - See “Guide to the new EventStore”
  - On the C3 software web page

- When we run code the obvious way to do it is interactively, from you own terminal window
- Fine for debugging, or running very short jobs
- But if we are running something longer we want to SHARE
- Therefore use BATCH QUEUES!
- On Linux machines a jobs is submitted via the command:

```
qsub -cell default -l arch=glinux myjob
```
Using ROOT

To run root from the command line use the new command:

```
c3root
```
(Thanks Chris!)

This ensures the you are using the correct version of root for your release at all times (even if you do c3rel).

There are three main ways to interact with root

1. Using the command line
2. Using GUI’s (many objects have GUI interfaces, e.g. TTreeViewer)
3. With scripts
Let’s look at the histogram file created in *RootExampleProc*

```c
root [0] TFile f("Xenu/root_example_data31.root")
root [1] f.ls()
TFile**         Xenu/root_example_data31.root
TFile*         Xenu/root_example_data31.root
  KEY: TH1F     h1;1    D^{0} #rightarrow K #pi #pi^{0} M_{bc}
  KEY: TTree    rtexnt;1        Root Example Ntuple

root [2] TH1F* hist = (TH1F*)f.Get("h1")
root [3] hist->GetEntries()
  (const Stat_t)1.09410000000000000e+04
root [4] hist->SetFillColor(30)
root [5] hist->Draw()
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root [6] gPad->Print("hist.eps","eps")
Info in <TCanvas::Print>: eps file hist.eps has been created
```
Notice the Title which has Greek letters

Root has a "kind" of LaTeX structure built in ... \pi, \rightarrow, etc ...

Root remembers all your commands ... ~/.root_hist
To repeat you can up arrow or look in this file
It’s often important to compare two distributions. We can either overlay the distributions, or stack them. Let’s use a script!

myScript.C

From the ROOT command line:
.x myScript.C

```c
{ ...
  TH1F *hum = new TH1F("hum", "Shower Energy", 80, 0, 0.8 );
  TH1F *hm = new TH1F("hm", "Shower Energy", 80, 0, 0.8 );
  ...
  TCanvas *c1 = new TCanvas( "c1", "Plots ", 1000, 500 );
  c1->SetFillColor(10);
  c1->Divide(2,1);
  c1->cd(1);
  hum->SetFillStyle(3001);
  hum->SetFillColor(kRed);
  hum->GetXaxis()->SetTitle("E_{shower} (GeV)层面的射程");
  hum->GetXaxis()->CenterTitle();
  hum->GetXaxis()->SetTitleOffset(1.2);
  hm->SetFillStyle(3001);
  hm->SetFillColor(kBlue);
  hum->Draw();
  hm->Draw("same");
  gPad->Print("hist_overlay.eps", "eps");
  c1->cd(2);
  THStack *hs = new THStack( "hs", "Shower Energy" );
  hs->Add(hum);
  hs->Add(hm);
  hs->Draw();
  hs->GetXaxis()->SetTitle("E_{shower} (GeV)层面的射程");
  hs->GetXaxis()->CenterTitle();
  hs->GetXaxis()->SetTitleOffset(1.2);
  gPad->Print("hist_stack.eps", "eps");
}
```
The two distributions are plotted one on top of the other. The two distributions are summed. The contributions are shown in different colours.
Don’t Forget to Explore the Menus!

Contextual Menus

Menus provide interface to member functions of objects
Using TTrees

• We get TTrees from a file that same way we get histograms

```
root [0] TFile f("Xenu/root_example_data31.root")
root [1] f.ls()
TFile** Xenu/root_example_data31.root
tf Xenu/root_example_data31.root
  KEY: TH1F h1;1 D^0 \rightarrow K \pi \pi^0 M_{bc}
  KEY: TTree rtexnt;1 Root Example Ntuple
root [2] TTree *tree = (TTree*)f.Get("rtexnt")
```

• To draw variables stored in the TTree object use Draw("var_name").
• To draw 2D or 3D separate variable names with a colon, i.e.,
  Draw("v1:v2") or Draw("v1:v2:v3")

```
root [3] tree->Draw("td0")
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root [4] tree->Draw("td0:tz0")
```
The Results

On its own ROOT doesn’t choose very good binning, or necessarily the style we would like!
Making Histograms of TTree Variables

• To make a histogram from a TTree variable we pipe the output of the Draw() to the appropriate histogram object.

```
TH1F *hist = new TH1F( "h1", "Track d0 (m)", 50, -0.01, 0.01 );
tree.Draw( "td0 >> h1" );
hist->SetFillColor( 42 );
hist->GetXaxis()->SetLabelSize(0.03);
hist->Draw();
```

• The same can be done for 2D plots ... again allowing much greater plotting flexibility.

```
TH2F *scatter = new TH2F( "h2", "Track d0 (m) vs z0 (m)",
                          20, -0.05, 0.05, 20, -0.005, 0.005 );
tree.Draw( "td0:tz0 >> h2" );
scatter->SetFillColor(30);
scatter->GetXaxis()->SetTitle( "z0 (m)" );
scatter->GetXaxis()->CenterTitle();
scatter->GetXaxis()->SetTitleOffset(1.2);
scatter->GetYaxis()->SetTitle( "d0 (m)" );
scatter->GetYaxis()->CenterTitle();
scatter->GetYaxis()->SetTitleOffset(1.3);
scatter->Draw("box");
```
We now have more appropriate binning and we can colour and set the plot style as we choose!

There are many other style for 2D plots, e.g., lego.
What are Cuts?

- We mentioned these briefly earlier ...
- A cut is simply a restriction placed on a variable, for example:
  - I want to plot track momentum, but only for tracks which come from the origin
  - I can cut on $d0$ and $z0$, i.e., plot track momentum only for tracks with $|d0| < 5\text{mm}$ and $|z0| < 5\text{cm}$.

```c
TH1F *htp = new TH1F( "h3", "Track Momentum (GeV), |d0| < 5mm, |z0| < 5cm", 50, 0.0, 1.5 );

tree.Draw( "tp >> h3", "(td0>-0.005 && td0<0.005) && (tz0>-0.005 && tz0<0.005)" );

htp->SetFillColor( 38 );
htp->GetXaxis()->SetTitle("p_{track} (GeV)" );
htp->GetXaxis()->CenterTitle();
htp->GetXaxis()->SetTitleOffset(1.2);
htp->Draw();
```

This is one way to make cuts in ROOT! OR! There is also a TCut object:

```c
TCut cut1 = "td0 < 0.005 && td0 >-0.005"
```
The Results

Notice that the distributions are different in shape AND notice that the number of entries is MUCH smaller for the histogram with cuts on d0 and z0.
Combinations

- Often we don’t want to plot exactly what we stored in our TTree, but various functions of one or more of the variables.

```cpp
TH1F *hist = new TH1F( "h1", "Track cos#theta", 50, -1.0, 1.0 );
tree.Draw( "tpz/tp >> h1", "tp < 2" );
hist->SetFillColor( 38 );
hist->Draw();
```

- ROOT also has a set of built in functions you can use, such as sin, cos, exp, etc.

```cpp
TH1F *ht = new TH1F( "h2", "Track #theta", 50, 0.0, 3.2 );
tree.Draw( "acos(tpz/tp) >> h2", "tp < 2" );
ht->SetFillColor( 40 );
ht->Draw();
```
Angular distribution of tracks from momentum variables stored in TTree.
Finally we often want to loop over Trees and do lots of complicated things at once
- We might want to sort the tree data and make many different histograms
- Or we might want to be flexible and plot many different variables
- Or we might want to make complicated functions of the tree variables
- For any of this we need more than just a simple script and we probably want to load in ALL the branches of our tree

- **TTree has a function called MakeClass(“ClassName”)**
- This makes a class which will setup all the branches and loop over your tree, all you need to do is fill in what goes in the loop!
- Also often we want to chain TTrees together, e.g., I have separate trees for data31, data32, etc.

- **Use the class TChain, which has function Add( TTree addedTree )** and inherits from TTree
Summary

• We’ve looked at some ways to fill histograms and trees in SUEZ

• We looked very briefly at running on data in the EventStore and submitting jobs to queues ... for more on this


• We saw how to use ROOT to look at histograms and trees, to make plots in various styles and print to files.

• But! there is still much more to learn and there are many resources to do so!
Matt Shepherd’s lunch talk


Jim Alexander’s Cleo101 from last year (day 2)


Official ROOT documentation at CERN (User’s Guide, How To’s, Forums, etc)

http://root.cern.ch/root/

My example code and scripts

/home/nea9/Source/RootExampleProc
/home/nea9/Source/scripts/RootExampleProc
/home/nea9/Source/scripts/RootExampleProc/root_scripts

Source Code

SUEZ tcl script

ROOT scripts

34
Finally - What to Make Sure You Know!

How to fill histograms and ntuples in SUEZ
How to use tcl scripts to run on EventStore
How to submit jobs to the batch queues
How to load histograms and trees from root files
How to run scripts and load classes
How to plot and manipulate histograms
  Set styles, fill colour, line colour, error bars
  Overlay and stack
  Normalize and scale
  Manipulate headings, legends, statistics
How to use the data stored in TTrees
  Make 1D and 2D histograms
  Use cuts and manipulate variables
  Get Branch variables into the current scope
  Loop over stored arrays
  Loop over the whole TTree
  Chain TTTrees together using TChain

Some of these points we’ve covered and some we haven’t. BUT you should still make it your goal to learn them all!