MOAN: An Example of How to do a ROOT-based DØ Analysis

Roger Moore

Michigan State University

Analysis with ROOT

- Why use ROOT for analysis?
  - Interactive: no 30 minute recompiles of D0 executables
  - Fast: highly optimised I/O means that ROOT jobs are I/O and not CPU limited
  - Standard Environment: No need for entire D0 environment to look at a file if you have compiled the libraries
  - Probably need to learn ROOT to show plots in any case
  - Excellent support
- ...but it's not all good
  - Not the worlds most stable program!
  - Uses interpreted "C++" which has subtle (and irritating) differences from real C++
  - C++ is not designed as a scripting language and it shows!

First Observations

- ROOT is missing one very useful PAW feature
  - Inability to plot functions as if part of the data
    - i.e. "nt/pl 1.yfunc%xfunc"
  - Very useful for rapid testing of ideas
- D0 has many data formats!
  - Thumbnail, RECO, custom physics groups...
  - It does not need another!
- ROOT's standard interface is buggy and very different from normal C++ (heavy F77 influence!)
- Conclusion:
  - Need a package which solves these problems

MOAN

- Authors: Jon Hays, Dave Evans and myself
- MOAN: Matched Object Analysis Network
  - Try using ROOT for a while and you'll see why we chose this!
- Design
  - Interchangeable use of pre-compiled and interpreted functions
  - Simplified interface optimized for typical analysis tasks
  - Expandable without needing to modify existing code, just adding new
    - Easy upgrading to new versions
  - Configurable from the command line...being added

Getting Started

- First you need to setup D0 CVS access and checkout the Moan package

```
> setup d0cvs
> cvs co Moan
cvs server: Updating Moan
U Moan/AUTHORS
U Moan/COPYING
...
```

- Now you also need to ensure that you have a valid D0RunII environment setup and that you have a more recent ROOT version setup

```
> setup D0RunII p13.08.00
> setup root v3_03_09a -q KCC_4_0:exception:opt:thread
```

Getting Started

- IMPORTANT: You need to ensure that your ROOT version is 3.03/09 (or maybe greater)
  - Bug fixes this includes are required
- Now need to setup autoconf environment

```
> autoheader
> automake -a
> autoconf
> automake
```

- Run the configure script to actually create the Makefiles
  - Must specify muo_cand version until package included in release

```
> ./configure -with-muo_cand=p13-br-03
```

Getting Started

- Can also supply CVS tags to configure for all D0 packages used by MOAN
  - Enables compatibility with any given ROOT file even if release has disappeared from disk
  - You have to map the release name to CVS tag though
- Configure script has several other options
  - Debug mode with --enable-debug
- Run configure with '--help' to see the full list
- Once configured, time to build the libraries...

```
> make
```

Getting Started

- Now you have built all the libraries for MOAN
  - All you need to do is load them into ROOT
  - N.B. You will need to change the paths shown below…

```
// Load MOAN Libraries
gSystem.Load("$ROOTSYS/lib/libPhysics.so");
gSystem.Load("Moan/thumbnail/TMBTreeClasses.so");
gSystem.Load("Moan/analysis/analysis.so");
gSystem.Load("Moan/cuts/cuts.so");
gSystem.Load("Moan/analysis/cuts/analysis-cuts.so");
gSystem.Load("Moan/analysis/tmb/analysis-tmb.so");
```

- This should load all the libraries you need to instantiate all the MOAN classes
  - Now you are ready to write an analysis…

Basic Concepts

- Analysis done using a tree-like structure of pre-compiled processor classes
  - Each processor performs a single, simple action
    - Matching
    - Extracting objects from input source
    - Filtering
    - Calculating invariant masses etc..
  - Pre-compiled classes required because ROOT interpreter cannot do virtual functions
- Macros assemble groups of processors into a framework to perform the actual analysis
  - Allows framework to change without recompilation
  - Eventually will add full set of methods to allow easy changing of the structure from the command line…not all there yet

Dataflow

- Processors pass lists of objects between themselves
  - Base object contains a 4-momentum and lists of matched objects
  - Processors can add a list of matches to any object using their name as a key
    - e.g. `obj.match("MuPlus")` will return the list of objects matched to 'obj' by the 'MuPlus' processor
- No new object formats needed if ROOT tree already has objects
  - ObjectInterface template provided
  - Inherits from given template argument and Moan's base physics object class
  - e.g. ObjectInterface<TMBMuon> has exactly the same interface as a TMBMuon class as well as that of a MOAN physics object

C++ Functions

- Need support for compiled and interpreted functions
  - Makes customization easy
- ROOT has no concept of function pointers
  - No support for interpreted virtual functions either!
- Solution: wrap all functions in pre-compiled classes
  - Provide classes for both compiled (C++) and interpreted (ROOT) functions
  - Templated but ROOT's use of templates requires prior knowledge of all instances
    - Adding more function classes easy but needs recompilation
- For interpreted functions this is handled automatically
  - Just provide the name of the function, MOAN will do the rest
- Compiled functions need to use the wrappers though

Input Formats

- Inputs can be from any source
  - To support a format you need a processor class that can extract objects from a file and feed them to the processor framework
- Several useful base classes provided to get access to data stored in ROOT Branches
- Currently code exists for d0analyze and Thumbnail ROOT formats
  - Only instantiate providers for the objects which you want
  - Avoids unpacking unwanted data and speeds up program...
- Unfortunately, due to a bug in ROOT, this does not work for Thumbnail ROOT files...yet
  - Currently unpacking only some of the data is a LOT slower than unpacking all of it!
  - Rene Brun & co aware of it, fix in next ROOT release

Making Plots

- All object processors (ones containing a list of objects) can have plots attached to them
  - Filled everytime the processor is run
- An intermediate DataAlgorithm class is used to provide maximum flexibility for plots
  - ObjectProperty: instantiated with a function which is run on every inout object
  - MatchProperty: instantiated with matched object list name and a function which is run on each pair of objects
  - EventProperty: instantiated with a function which is given the entire list of input objects for that event
  - Other properties available or write your own!

Making Plots

- Data Algorithms plot the return values of functions
  - e.g. a 'pT' function attached to an ObjectProperty will plot the pT of the output objects of the processor.
- Separate Plotter processor being written to support multiple input sources
  - e.g. nJets vs. nMuons
- 2D Plots supported
  - single value vs. multiple values
  - multiple values vs. multiple values if same number in each case
- Two different methods to add plots

```
add1DPlot(DataAlgorithm *,TH1 *)
add2DPlot(DataAlgorithm *,DataAlgorithm *,TH2 *)
```

Simple Example: Muon Isolation

- Simple example macro to plot the isolation of muons in a Thumbnail Tree file
- First we need to open the data file and create a histogram to fill
  - Standard ROOT, nothing new

```
// Open the data file
TFile *data=new TFile("MyDataFile.root");
// Create the ECone Isolation histogram
TH1 *muisoh1=new TH1F("MuIsoH","#mu Isolation",
                      0.,10.,50);
```

- Create a provider to extract the muons from the file

```
// Create a provider of thumbnail muons
ObjectProcessor *muons=new tmb::MuonProvider;
```

Simple Example: Muon Isolation

- *Now we need to define a function to calculate the muon's E-Cone Isolation*

```
// Calculate E-Cone Isolation of a Muon
// Defined as 0.4-0.15 cone energies
double eConeIso(void *dummy) {
  moan::ObjectInterface<TMBMuon> *mu=dummy;
  return mu->EInCone4()-mu->EInCone15();
}
```

- *Create an object property data algorithm for it and add the plot to the provider*

```
// Create the data algorithm for the isolation
DataAlgorithm *muisoalg=new ObjectProperty(eConeIso);
// Add the plot to the muon provider
muons->add1DPlot(muisoalg,muisoh1);
```

Simple Example: Muon Isolation

- *Finally we need a special Analyser processor*
  - *Processor which runs the processor framework attached to it over the data provided*

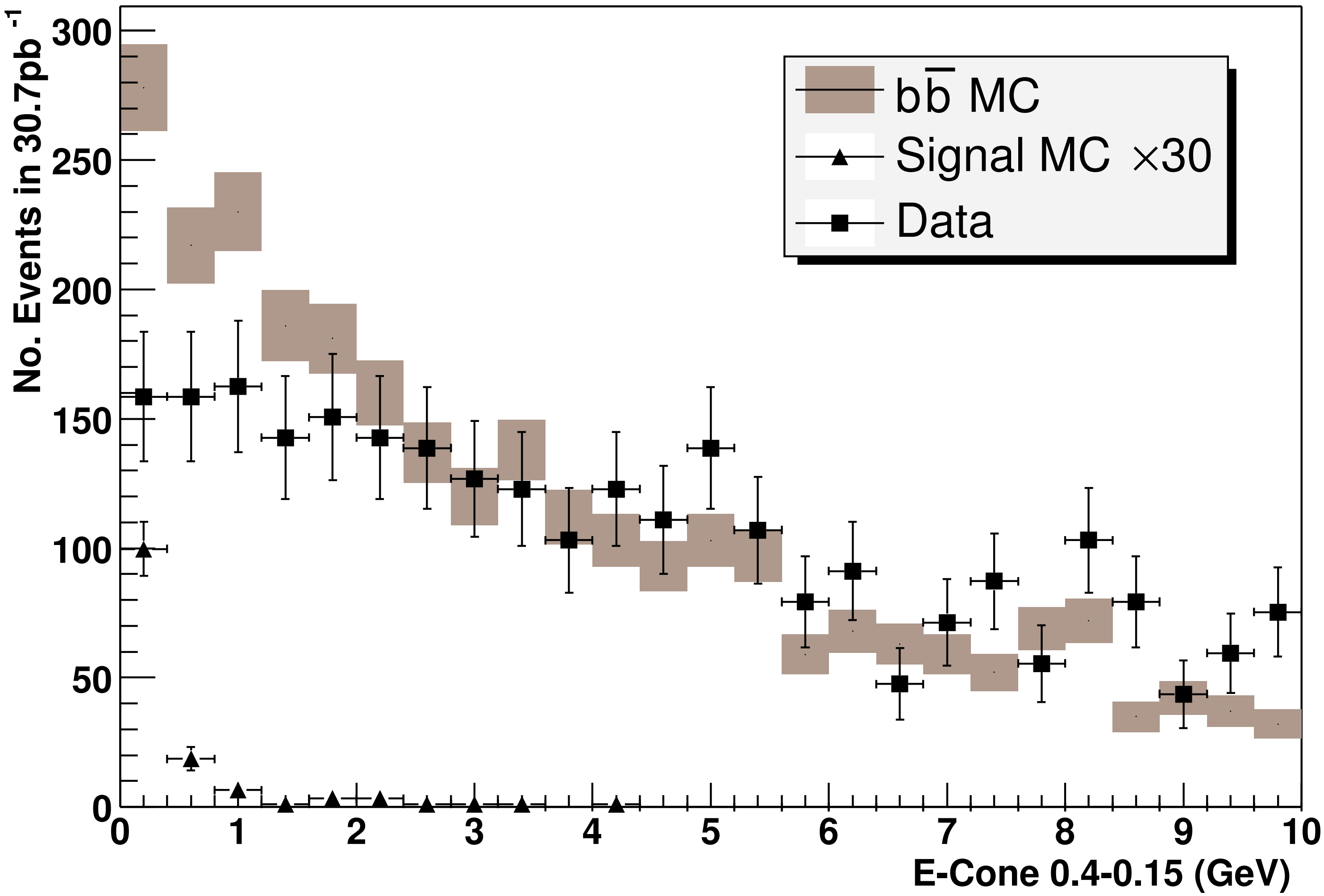
```
// Create the Analyser
moan::Analyser a("MuonIso",data);
a.addProcessor(muons); // Add the muon provider
```

- *Create an object property data algorithm for it and add the plot to the provider*
  - *At this point we just need to run the analysis*
- *Call the analyser execute method*
  - *Takes optional arguments to limit the number of events to process and to skip events*

```
a.execute() // Run analysis on all data
```

- *Histogram should now be filled...*

Simple Example: Muon Isolation



Anatomy of a Processor

- All processors are required to provide two methods
  - Constructor to set up options and attach to child processors
  - **`void doRun(void)`** which is called once (and only once) for each event that the processor must process
- Object processors, ones providing a list of objects as an output have the following methods
  - Constructor to set up options and attach to child processors just as for a normal Processor
  - **`void processData(void)`** which replaces the "doRun" method from the standard processor
  - **`void addObject(PhysicsObject *)`** which adds the given object to the output list
- Easy to write your own if you need to...

Event Selection

- Processors great for filtering objects but sometimes need to filter events
  - e.g. plot this only for events with 2 muons with pT> 5GeV/c
- MOAN provides two special classes to do this
  - Decider: alters flow of the processor network
  - Selector: examines input data and gives a 'true' or 'false'
- Decider has three special methods
  - **`void addSelector(Selector *)`**
  - **`void addPassProcessor(Processor *)`** adds a processor to run if the selectors all return 'true'
  - **`void addFailProcessor(Processor *)`** adds a processor to run if any selector returns false
- ObjectSelector to count input objects provided

Example: Dimuon Analysis

- More complex macro to select and plot invariant masses of different sign, isolated muons from a TMB ROOT file
- First create providers for muons and jets

```
ObjectProcessor *muon = new tmb::MuonProvider;
ObjectProcessor *jets = new tmb::JetProvider("JCCA");
```

- Now define the function to select "good" muons

```
// Returns 1 for a good muon
double goodMuon(void *dummy) {
  moan::ObjectInterface<TMBMuon> *mu=dummy;
  if(mu->nseg()!=3) return 0;
  if(mu->chisq()<0. || mu->chisq()>200.) return 0;
  return 1.;
}
```

Example: Dimuon Analysis

- *...and then another couple of functions to select positive and negatively charged muons*

```
// Return 1 for anti-muons (mu+)
double muPlus(void *dummy) {
  moan::ObjectInterface<TMBMuon> *mu=dummy;
  return mu->charge()>0. ? 1. : 0.;
}

// Return 1 for muons (mu-)
double muMinus(void *dummy) {
  moan::ObjectInterface<TMBMuon> *mu=dummy;
  return mu->charge()<0. ? 1. : 0.;
}
```

Example: Initial Framework

- *Now that we have the functions and providers we need to create filter processors and attach them*

```
ObjectProcessor *goodmu =
  new moan::FunctionFilter("GoodMu",muon,*goodmuon);
ObjectProcessor *mup =
  new moan::FunctionFilter("MuPlus",goodmu,*muplus);
ObjectProcessor *mum =
  new moan::FunctionFilter("MuMinus",goodmu,*muminus);
```

Example: Calculating Masses

- Now we can create a processor to calculate the invariant mass of the mu+/- pairs
  - Creates new 'mass' objects from lists of mu+ and mu-

```
ObjectProcessor *mumumass =
  new MassProcessor( "MuMuMass",mup,mum);
```

- ...and then a filter processor to select ones around the upsilon mass
  - Needs a new function also defined here

```
// Selects masses in range 8-12 GeV/c2
double upsilonMass(void *dummy) {
  PhysicsObject *obj=dummy;
  return (obj->p().M()>=8.0 && obj->p().M()<=12.0) ? 1. : 0.;
}

ObjectProcessor *upsmass =
  new moan::FunctionFilter("UpsilonMass",mumumass,upsilonMass);
```

Example: Writing out Events

- At this point we have a processor which has a list of all the mu+/- pair masses close to the upsilon
  - Suppose we want to now save these events?
- MOAN provides an output processor
  - Will write out the current event from the input tree if the input processor has the given number of objects (or more)
  - Limited to only writing out data for which providers are instantiated
    - ROOT limitation/bug, hopefully will be fixed...

```
Processor *upsoutput =
  new moan::OutputFilter("UpsilonOut",upsmass,1,
                         "upsilon.root");
```

Example: Adding Plots

- To add a plot first we need to create histograms…

```
TH1F *mumumh1 = new TH1F("MuMuMass",
  "#mu-#mu Mass",100, 0., 50.);
TH2F *mumumpth2 = new TH2F("MuMuPTvM",
  "#mu-#mu p_{T} vs. M", 60,0.,30., 100,0.,50.);
TH1F *upspth1 = new TH1F("UpsilonPT",
  "#Upsilon p_{T}",60, 0., 30.);
```

- Now we can add the plot using a data algorithms already supplied in the library to plot masses and pT of input objects

```
mumumass->add1DPlot(&oprop::M,mumumh1);
mumumass->add2DPlot(&oprop::pT,&oprop::M,mumumpth2);
upsmass->add1DPlot(&oprop::pT,upspth1);
```

Example: Running the Analysis

- *To actually run the analysis on a file we need to create a top level Analyser processor and give it the TTree or TChain to process*
  - *...and then add the top level processors that we need to run, in this case the upsilon output processor*

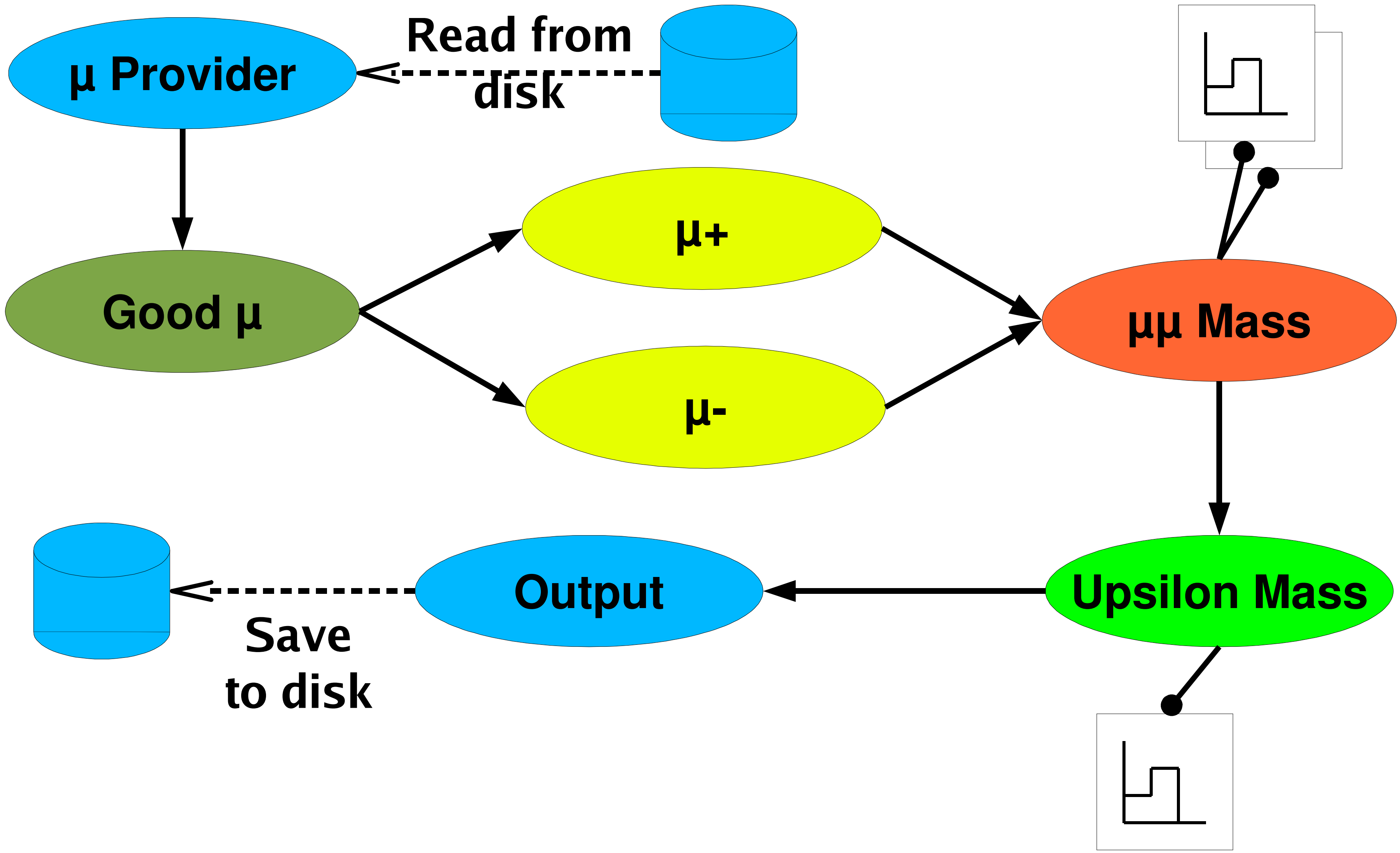
```
moan::Analyser a("Analysis",myTree);
a.addProcessor(upsoutput);
```

- *Finally, to run the analysis we just call the Analyser's execute method with an optional maximum number of events to process*

```
// Run over no more than 1000 events
a.execute(1000);
```

- *This will fill all the attached histograms*

Example: Final Framework



Muon Candidate

- Muon canditate integrated with a special provider
  - Select version to use with switch to the configure script
- Provider requires uses other providers to get data needed by algorithm

```
MuonCandProvider(MuonProvider *muons,
                 JetProvider *jets,
                 TrackProvider *tracks,
                 VertexProvider *vertices,
                 GlobalProvider *global);
```

- Output is list of MuoCandidate wrapped objects
  - Identical to official MuonID certified objects
  - "LocalCentral" muons only at the moment

How to Find out More...

- So far this tutorial is probably the best documentation for MOAN!
- However source code is carefully documented using doxygen-style comments
  - Best place to look is in the C++ header files: *.hpp
- Code is still changing and source code is the one place guarenteed to be uptodate!
- Current state is ~beta: most things work most of the time
  - Still need to be willing to look at C++ code but should not need to delve into the innards of ROOT!

Things To Do

- Lots of ideas for improvements...but not enough time to add them all!
  - Outputing lists of objects and their matches
    - Load back in a saved analysis and continue
  - Make interface to Harry M's D0 cuts package work...it compiles!
    - `moan::Filter *mufilt = new moan::Filter("LowMassMu",musrc,cutfunc::M<10.);`
  - More event level quantity support: more selectors
  - Utility macros to set up simple frameworks
  - Member function caller to avoid need to write functions and object properties for every method of a class
    - In and compiles, haven't had a chance to test it yet...
- People are welcome to jump in and add their own code
  - So far development driven by authors' analyses

Conclusions

- ROOT based analysis works
  - Being used for SUSY like-sign dimuon analysis and others
- Major advantage is speed
  - Rapid, interactive analysis
  - Fast: even with current bug takes ~30-50ms/event on 1GHz pentium III CPU
    - When bug fixed expect faster processing
- Would be nice if D0 could agree on ONE format for ROOT that everyone could use
  - d0analyze, TMBTree, custom physics group formats all in mix
  - Common object interface hierarchy would be EXTREMELY useful
  - ...but needs lot of agreement and common effort