How to Use the MakeGoodPlot Tools

Geoffrey Smith

University of Notre Dame CMS Group

UNIVERSITY OF NOTRE DAME

College of Science

Overview

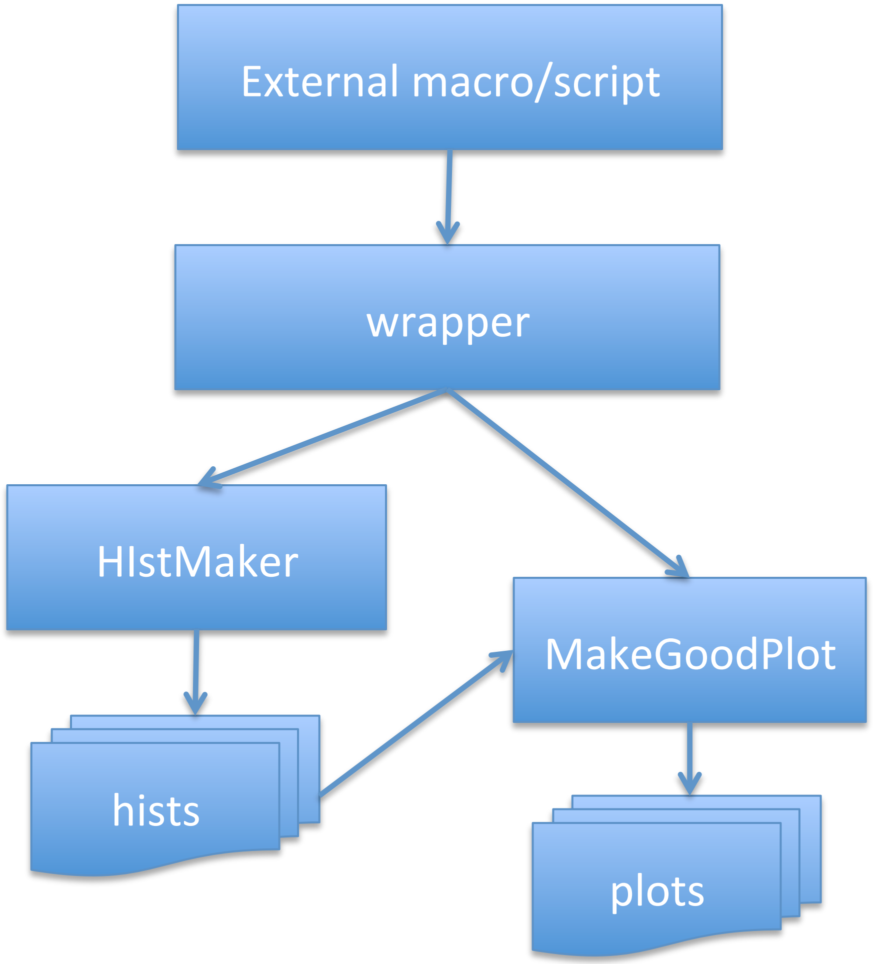
- These slides describe how to use the histogram- and plot-making code for the multilepton EFT analysis.
  - Note: if you need to do something with the trees other than make plots (such as train a BDT), you will need to use different code!
- Code on github here.
- There have been several versions of this code, with the current version (“v2”) being a complete rewrite of the code ported from the multilepton ttH analysis in late 2017.

Overview II

- This version was re-written with the specific goal of being able to produce large numbers of plots for the EFT analysis as quickly and efficiently as possible
  - Modular design for assembly-line style production. Goal: make it as trivial as possible to add/remove plots or add new code for producing plots
  - Option to use multithreading to speed up filling of histograms
  - Histogram-making and plot-making are separated; i.e., one can remake plots quickly without having to run the whole chain
  - Helper classes/functions for applying analysis weights, assembling hists into plots, posting plots to web area, etc.
- Code is written in c++, and uses ROOT.

Structure of code

- Everything contained in wrapper.h
  - Defines basic workflow for both histo- and plot-making
  - Depending on how wrapper function is called, histo- and plot-making can be run together or separately
- Wrapper function typically called from ROOT macro or shell script which contains the list of samples



Calling wrapper function (wrapper.h)

- `void wrapper(std::vector<int> samples, int mode=2)`
  - mode=1 -> (re)make hists only
  - mode=2 -> (re)make plots only
  - mode=3 -> (re)make both hists + plots
  - Takes std vector corresponding to samples to run over. Typically this is supplied when wrapper function called from a ROOT macro -- see makegoodplot.C for an example of how to do this.
  - Sample-to-int assignment defined in loadsample.h
- Function is also overloaded in wrapper.C to take only a single int. This is mainly for when the list of samples is defined in a shell script instead of a c++ macro, i.e. for batch running. See makegoodplot.sh or makegoodplot_local.sh for usage examples.
- Other versions of wrapper function exist (e.g. wrapper_lobster.C). See later slides for use cases
- Recipes for running scripts also given in later slides

Inside wrapper.h

- Function written primarily with the requirements/ restrictions of the TTreeProcessorMP ROOT class in mind
  - Provides multithreading functionality
  - Based off of example given in $ROOTSYS/tutorials/ multicore/mp102_readNtuplesFillHistosAndFit.C
- Main restriction: unit of “work” sent to each thread is in form of c++ lambda function
  - Can take only a single argument: a TTreeReader (this is the chuck of tree being processed by a given thread), and must return a TObjArray.

Note on TObjArrays

- The TObjArray returned by HistMaker is exactly that – a container holding TObjects. Since all ROOT objects inherit from TObject, any ROOT object (not just histograms) can be stored in a TObjArray.
- The code makes extensive use of TObjArrays and other containers to perform operations on large numbers of TObjects at a time (histograms, TCanvases, etc.). This is the “assembly-line” aspect of the code.
- The histograms can be accessed/manipulated directly from the TObjArray:

```
auto hist = (TH1D*)objarray.FindObject(“hist_name”);
hist->Draw();
```

Inside wrapper.h II

- For a given sample, the list of files to be run over is obtained with the help of the loadsample function, and is then passed to a TChain via the AddFileInfoList method
  - To avoid various i/o and memory issues, the maximum number of files to consider at a given time is set by the "filesatatime" variable.
- The histo-making jobs are then run by calling `workers.Process(…)`
  - Runs the workflow specified in "workitem" in a number of parallel threads
  - Combines the results from all the threads in the form of a TObjArray which contains the output histograms.
- The histograms are saved to a .root file, along with the special "NumInitialWeightedMCevents" histogram obtained separately with a dedicated helper function.
- If specified (i.e. by setting mode=2 or 3), the MakeGoodPlot code is then run. Note that this step does not use multithreading. A MakeGoodPlot object is instantiated, which loads all histograms from the files produced in the previous step. Plots are made from these histos by calling one of the drawAll* member functions of MakeGoodPlot.

Histogram-making workflow (contents of workItem in wrapper.h)

- `histmaker->setBranchAddresses(reader);`
  - See setbranchaddresses.h. This is where we associate the objects used to access information about muons, electrons, etc. with the names of tree branches. This can change if the format of the information being stored in the trees changes, or if we want to change which branches are being accessed.
- `histmaker->bookHistos();`
  - Histograms to be filled later must be booked here (see bookhistos.h). Each must have a unique name. They are added to a std map which can be accessed from any other member of the HistMaker class. For example, to fill a given TH1D, one would call `th1d["some_hist"]->Fill(…)`, where some_hist is the name of the histogram specified in bookHistos.

Histogram-making workflow (contents of workItem) II

- `histmaker->run(reader);`
  - Runs the event loop (code at right)
  - The actual filling of histograms is done inside doOneEvent()
  - See run.h + dooneevent.h
- `histmaker->collectResults();`
  - After the histos have been filled, this takes the std map of histograms, copies it to a TObjArray, and returns the TObjArray

```
void HistMaker::doOneEvent()
{
     // Whatever code you want
     // to run for each event
     // (fill hists)
}
void HistMaker::run(TTreeReader
& newreader)
{
     while (newreader.Next())
     {
          doOneEvent();
     }
}
```

Plots from histograms

- MakeGoodPlot class does 3 things:
  - Loads the histograms from .root files produced by HistMaker into TObjArrays
  - Provides variety of helper tools to make plots from the hists contained in these TObjArrays
  - Collects all the plots (TCanvases) into another TObjArray and draws them all at once in the manner specified by the user (i.e., save plots to file or draw to screen)

Plot-making workflow

- Using the `MakeGoodPlot(std::vector<int> thesamps)` constructor will load the hists and initialize some member variables to default values (see setup.h and rateinfo.h)
- All the user has to do is then call one of the drawing functions to draw all plots at once
  - newplots->drawAllToScreen() will draw all TCanvases directly to the screen in the usual x-windows
  - newplots->drawAllToFile("plttest","pdf”) will dump all plots to file in the specified format. If “pdf” or “root” is specified, all the plots are saved to a single pdf or root file. If “png” is given, the plots are saved as separate png images.
  - newplots->drawAllToWebArea(“name_for_these_plots”,”png”) does the same as drawAllToFile except the files are saved to the user’s web area in the specified directory instead of the local directory.

Plot-making workflow II

- All of the drawAll* functions are just wrappers around drawAll(), where users should include code that does the actual creation of the plots (see drawAll.h)
- Minimal steps that need to be done in this included code:
  - Create a new TCanvas to contain your plot
  - Grab whatever hist(s) you need from the histogram TObjArray (in the way described earlier)
  - Do something with the hists, and draw them to the TCanvas
  - Finally, you must add your TCanvas to the array of TCanvases: `canvas.Add(mycan);`
- There is a helper class available (called GoodPlot) that attempts to automate many of the typical things one would normally do in root to make decent-looking (or hopefully approval-quality) plots
- See lepeff_plots.h or jetcleaningstudies_plots.h for some basic examples, and standardplots.h for more advanced examples using GoodPlot class

Notes on included code

- All external code (c++ libraries, ROOT, CMSSW), various utilities/helper functions, and basic components of the HistMaker and MakeGoodPlot code is included in includes.h
- Only things NOT included in includes.h:
  - New histogram-making functions should be included at the top of dooneevent.h
    - Don’t forget to also add new hist-making member functions inside HistMaker class declaration (HistMaker.h)
  - New plot-making functions should be included at the top of drawall.h
    - Don’t forget to also add new plot-making member functions inside MakeGoodPlot class declaration (MakeGoodPlot.h)

Various recipes

The following assumes access to/knowledge of the NDT3 or the ndpcs at CERN, and knowledge of condor and/or Lobster.

How-To: run interactive session

UNIVERSITY OF NOTRE DAME

- On earth, an interactive recipe would be as follows:
  - cd to MakeGoodPlot2 dir, cmsenv
  - If not already done, edit loadsample.h to point to the trees you want to run over (most likely on hadoop)
  - Edit makegoodplot_local.sh to include the samples you want
  - If needed, edit wrapper.h. I found the following values to be reasonable when running on earth:
    - njobs = 24
    - filesatatime = 2000
  - Note: please use good judgment when running with multiple cores on earth, as this may impact the work of other users.
  - Finally, do: `./makegoodplot_local.sh`
- After this finishes, before running the plot-making step, you will need to hadd the hist files:
  - `./haddhists.sh`
- Then, after checking that makegoodplot.C has the samples you want:
  - `root –l makegoodplot.C+`

How-To: run interactive session II

UNIVERSITY OF NOTRE DAME

- If your samples are located on one of the ndpc disks, it's possible to run the hist-making code on the ndpcs
- The steps are the same as on earth, except in wrapper.h:
  - njobs = ~8-12 (16 max)
  - filesatatime = 2000? (not so much of an issue on ndpcs)
- As of this writing, will probably need to fiddle with parameters in loadsample.h – the "if (!atND)" section at the bottom will most likely not have the desired behavior
- The plot-making portion of the code should run exactly the same as on earth

How-To: run condor batch session

- It's possible to run the histo-making portion of the code using condor
  - You will first need to un-comment the last ~4 lines of the HistMaker default constructor, and comment out the previous ~4 lines.
  - Edit makegoodplot.sh to include the samples you want
  - If needed, edit makegoodplot.submit (this is the condor submit file)
  - Type: `condor_submit makegoodplot.submit`
- Use the usual condor commands to see how the jobs progress. You will then need to run the plot-making portion of the code interactively.
- Note: I have found that using Lobster (see next slides) is preferable to trying to use bare condor on the NDT3

How to run Lobster batch session

- The histogram making portion of the code has been modified to be able to run with Lobster
  - The instructions for running the plot-making portion of the code (after you have made the histograms) is pretty much the same. The only thing you need to change is in the instantiation of the MakeGoodPlot class. See the bottom of makegoodplot.C for details.
- These slides do not attempt to give a tutorial on how to install/use Lobster. For documentation on Lobster please see here: http://lobster.readthedocs.io/en/latest/

How to run with Lobster (II)

- After installing and setting up your Lobster environment, please take note of the following files in the MakeGoodPlot2 directory:
  - wrapper_lobster.C
  - wrapper_lobster.py
  - config_lobster.py
- config_lobster.py is the Lobster config. To start a lobster process type:
  - `lobster process config_lobster.py`
- You can then use the usual lobster commands to monitor the progress of the jobs, etc.

Inside config_lobster.py

- The command that Lobster runs is the following:
  - `python wrapper_lobster.py label inputfiles`

  where `label` is a string that determines the sample you're running over, and `inputfiles` is a subset of root files for that sample.
- You most likely will only need to touch the first part of the config_lobster.py, where input and output directories are defined.
- You can include/not include samples by uncommenting/commenting lines that append strings to `mysamples`

Inside wrapper_lobster.py and wrapper_lobster.C

- wrapper_lobster.py just calls wrapper_lobster.C
- wrapper_lobster.C is a simplified version of wrapper.h
  - Doesn't use any multithreading (not necessary with Lobster)
  - Just fills the histograms using the HistMaker class, and saves them in the same way as in an interactive session, except here the output file is always just "output.root," and Lobster saves it to a directory based on the sample.