VqqHcc Analysis Manual : A Guide through the Code

Peter Young

June 5, 2025

Contents

Chapter 1

Introduction

So you want to get involved in the VqqHcc analysis at CMS started by D. Nguyen and P. Young? Well, you've come to the right place. This documentation is a great working guide for how to get started and what needs to be done. Follow this guide and you should be able to run the code yourself.

1.1 Resources

There are many resources you will want to keep track of including documentation with CMS, CERN, and this analysis. All of these types of documents and their links can be found in the bibliography at the end of this guide. We will reference them frequently, so it's a good idea to bookmark them on your browser.

In terms of browsers, I use Google Chrome. However, any browser should work for you since you don't need to be connected to your grid certificate through a browser for this (typically done with Firefox) [1].

Chapter 2

Basic Setup

What is the most basic information you need to know for running the code for this analysis? As previously stated in Chapter 1, all the resources needed are listed in the bibliography. It's best to start with how to access the code.

2.1 Setting Up Your Environment

This assumes that you already have an account setup on LPC and know how to navigate it. We will skip all of those steps. Once we are on LPC, we need to get the proper version of the CMSSW software. While we are using (at this point) a very old version of NanoAOD - specifically `NanoAODv9`, a lot of the analysis software used needs a much newer version of CMSSW. For this analysis, we currently use `CMSSW_14_0_6`. We first need to make sure we set up this version of CMSSW.

```
cd [your_working_space]
csmrel CMSSW_14_0_6
cd CMSSW_14_0_6/src
cmsenv
```

Once this area is setup, you can download the code from the Github:

```
git clone https://github.com/peteryouBuffalo/VHccAnalysis.git
cd VHccAnalysis
```

Note that this repository is public, so you should not need any special permissions to do this. However, if you wish to push code to Github at any point, you will need to be given access.

This is the basic environment you need and you should be able to access all the code!

2.2 Environment for HiggsCombine

There are many newer tools taken from other analyses that we use, specifically HIG-24-017, that use newer versions. Thus, to properly setup a space for HiggsCombine, you need to do the following:

```
cd [your_working_space_v2]
cmsrel CMSSW_14_1_0_pre4
cd CMSSW_14_1_0_pre4
cmsenv
```

Note that you likely want to keep two workspaces - one for the analysis and its actual work, and a second one for the fitting/Combine steps. Likely, you want two spaces named as something like what follows:

- `/uscms_data/d3/[user]/boosted_VHccAnalysis/Ana`
- `/uscms_data/d3/[user]/boosted_VHccAnalysis/Fit`

Depending on how savvy you are with writing code, it might be helpful to write yourself special terminal commands to easily switch back and forth between the two, i.e. `mv_ana` and `mv_fit`.

The fitting and Combine code scripts are kept in a separate branch within the code. To access this code, in your `Fit` working space, you need to do the following:

```
git clone https://github.com/peteryouBuffalo/VHccAnalysis.git
cd VHccAnalysis
git checkout Fitting_scripts
```

The branch `Fitting_scripts` contains all of the necessary scripts for the Fitting process.

Now that you've got all the workspaces set up, it's time to dive into the code!

Chapter 3

Analysis Code

At the time this is written (due to debugging and fixes in code), there is likely other branches where code is stored. However, by the time you read this, likely everything you need is just in the main branch of the Github repository [2]. It's good for us to go through step-by-step how the code is setup.

3.1 Overview & Structure

The code structure is setup and based on the `EDMAnalyzer` structure created by CMS [3]. It is not necessary to understand how to build this yourself, and you can just start with the base that we have setup. There are two major components to the code that you initially need to understand.

- `src` - this is the source repository where the major code for the analysis is stored.
- `Ana.cxx` - this is the overall C++ script that compiles everything into the analyzer and runs the actual analysis via the `EDMAnalyzer` structure.

- `UtilScripts` - this is a directory of files generally related to calculating special transfer functions, efficiencies, calibrations, etc.
- `macros` - this is a directory for many macro scripts that do stuff like plotting variables we're interested in. Stuff tends to get mixed between `UtilScripts` and here.
- `SubmitToCondor` - this contains scripts needed for submitting jobs to Condor to run on the grid.

There are other folders that are mostly just storage:

- `CalibData` - this contains locally-stored calibration files
- `Configs` - this contains files related to constants, and configuration information related to ROOT files, samples, cross sections, etc.
- `Dataset_lists` - this contains files to be used for associating samples with the appropriate NanoAOD files
- `FileLists` - this contains the actual xrootd sources for where to find the files stored in the CERN computing system.

3.2 Source Folder, src

Every file in this folder has its use. However, there are many that are there for general structure and can be ignored. Any file not mentioned in this section can be ignored for your general purposes.

3.2.1 Global.cxx/.h

The `Global.cxx/.h` files take global constants stored in `Configs/inputParameters.txt` and properly stores them so that they can be used and called anywhere in the code. While it is inefficient, the parameter must be listed in the text file

and then set up in `Global.h` so that it is recognized. Otherwise, it will just give null values. These values can be referenced in the following manner:

```
CUTS.Get<T>("constants_name")
```

T is the type of variable that the constant is. For example, the p_T cut for lepton vetoing is given by:

```
CUTS.Get<float>("lep_veto_pt")
```

Unless a new constant is added, you likely will not have to modify this file. You can change the value of a constant in `inputParameters.txt` without needing to modify these scripts.

3.2.2 Obj.cxx

At their lowest level, the particles we store are essentially just momentum four-vectors. ROOT has a class for these (TLorentzVector), but we want to be able to build upon them to make easier classes. The `Obj.cxx` file contains a multitude of classes that build upon the four-vector to store proper information. They are as follows:

- `LepObj` - leptons (also general use)
- `JetObj` - AK4 jets
- `JetObjBoosted` - AK8 jets
- `ZObj` - Z boson
- `HObj` - Higgs boson

For what specific variables are stored for each type, you can check the file more closely. The most useful element is each object has a TLorentzVector `m_lvec` which has its own functions for pulling desired values. Each object is constructed from the appropriate information - leptons from general information, jets are similar, and Z and Higgs bosons are reconstructed from jet objects.

3.2.3 Plots.cxx/.h

There will be lots of independent plots/histograms in the analysis that just use `TH1*` from ROOT. However, there are many cases where things would get redundant and complicated very quickly, so we have groups of plots that are produced together to make things simpler. Within `Plots.h`, we define many classes of plots that simplify code. Note, each of these classes are designed to allow for events to be weighted and have an optional parameter for weight (default = 1). All plots are also initially set up so that the values have their uncertainty stored as w^2 (i.e. `TH1*->Sumw2()`). Note that there are many ways in which these classes can be improved and restructured so save redundancy, but it is not useful at this time since everything works.

VHPlots

`VHPlots` is a class that contains plots for expected events of this analysis - ones containing a Higgs boson produced in association with a vector boson. Thus, any plots we could want related to such things area included here - p_T, η, mass, phase-space separation, etc. This plot takes in the information of bosons by taking in a `HObj` instance and `ZObj` instance. Additionally, information about jets used in the event can be filled from a vector of jets. For example, this would look like:

```
HObj h_obj(//...);
ZObj z_obj(//...);
std::vector<JetObj>& jets {//...};
```

```
vh_plots->Fill(h_obj, z_obj);
vh_plots->FillJets(jets);
vh_plots->FillNjet(jets.size());
```

The whole of the list of plots in the class is returned in a vector by the method `returnHisto`.

VHBoostedPlots

This is the same thing as `VHPlots` but with extra information included for the boosted case.

HBoostedPlots

This is a class that could be used with the above to help save time of information on a boosted Higgs object. It stores the general information for a boosted Higgs object.

BoostedJetEffPlots

This is used for 3D plots related to calculating efficiency. You can ignore this for now.

EffPlots

This is a class for helping calculate efficiency. You can also ignore this for now.

3.2.4 JESUncPlots

This is a class created for purposes of calculating and storing JES uncertainty information.

3.2.5 Reader.cxx/.h

While we have access to all the branches stored in `NANOAODv9` by default, we still need to set up our code to read and recognize these branches from the ROOT tree where they're stored. `Reader.h` is where you set up branches to be read to be used in the analysis. Some branches only exist in MC or certain years, so make sure to set it up properly so that the branches only exist where needed. For example, we set up macros like `DATA_2016`, `MC_2016`, etc. to differentiate between each year and data vs MC. These branches for input all take one of the following general formats:

```
TTreeReaderValue<T> var_name = {fReader, "string_name"};
TTreeReaderArray<T> var_name = {fReader, "string_name"};
```

The first is used for variables where there's only one value for the entire event. The second is used where there are multiple in a single event. For example, the following are included for jets:

```
TTreeReaderValue<UInt_t> nJet = {fReader, "nJet"};
TTreeReaderArray<Float_t> Jet_pt = {fReader, "Jet_pt"};
```

These names must follow from what's stored in `NANOAODv9` given by the NANOAOD documentation [4].

3.2.6 Selector.cxx/.h

These contain many special methods (we create/add) for special calculations needed, such as uncertainties, calibrations, etc. Unless you're looking for a

specific method or need to update one, there's no need to look in here.

3.2.7 VbbHcc_selector.cxx/.h

This is the **MAIN** file you'll be interested in. This is where the main analysis is done. This will be discussed later in detail.

3.3 Ana.cxx

All the code in the `src` folder is fine on its own, but it needs to be used. This is where `Ana.cxx` comes in. This is the main entry point of the code running. It properly sets up the processor and instance of the `VbbHcc_selector`. If any special information such as proper files per year or calibarations must be added, they are included here based on the proper year/conditions. There is very little time in which you will need to modify this script.

Chapter 4

Main Event Selection Code

To understand how to use and modify the code, you need to know what goes where inside the `VbbHcc_selector` code. This will not go over every single element of the analysis. Duong and I can explain it (or you look through the analysis notes). Instead, this will just tell you what has to go where.

4.1 Header File

Inside the header file, you need to declare all the histograms and plots you're interested in. This can either be `TH1D` instances or instances of our specialized plot classes. Examples of what this looks like is:

```
TH2D* h_Xcc_vs_pQCD_raw;
VHBoostedPlots* h_ZccHcc_PN_med;
```

Nothing fancy is really needed here. Now, we need to look inside the C++ file.

4.2 SlaveBegin

The first section of the selection code is the `SlaveBegin` which takes the initial reader and allows us to set things up. We need to define all the plots and classes we declared in the header file. Once all the proper elements are defined, they need to be added to the reader's output list so they can be accessed in the output files. This is done as follows:

```
r->GetOutputList()->Add(plot_name);
```

For the special classes we have, this is done by looping through all the histograms returned by that class's `returnHisto()` method, as seen here:

```
std::vector<TH1*> tmp = h_ZccHcc_PN_med->returnHisto() ;
for(size_t i=0;i<tmp.size();i++) r->GetOutputList()->Add(tmp[i]);
```

4.3 Special Methods

There are special methods that exist in our code. For example, for generator-level particles, we want to get the IDs and information of Z and H decay daughter particles.

4.4 Process

`process` is where the main processing occurs. This breaks down into mulltiple steps.

4.4.1 Event Weights

First, the weights are calculated given either values by default inside of the ROOT files or methods/calibration files we have stored.

4.4.2 Object Selection

Given the criteria we use for this analysis defined in the analysis notes, we go through the information stored in NanoAOD to choose only leptons and jets that we want.

4.4.3 Control & Signal Regions

Once all objects are selected, we go through the regions of the analysis that we want. This includes:

- W-tag CR
- SR - ZccHcc
- CR - QCD (ZccHcc)
- CR - Top (ZccHcc)
- SR - VqqHcc
- CR - QCD (VqqHcc)
- CR - Top (VqqHcc)
- CR - Top, QCD-enriched (VqqHcc)

4.5 Terminate

This is where you should delete or close any necessary items that we need to be closed. Currently, we do nothing here, so you can ignore it for the most part.

Chapter 5

Running the Code

There are many steps to running the main analysis. The first thing you want to do is run over the ROOT files to get the output histograms that we desire. This is the main per-event code run by `Ana.cxx`. Once this is done, further analysis using macros can be done.

5.1 Compiling the Code

The code must be compiled for the different eras and conditions that are desired. There are makefiles that simplify this for us. The general format is as follows:

```
make FORMAT=[DATA_MC_YEAR]SUBFORMAT=[SUB_CATEGORY] \
SAMPLE=MC_NOTVZ PROCESSING=NORMAL NANOAOD=NANOAODV9 INPUT=TCHAIN
```

The main parts to focus on are the `FORMAT` and `SUBFORMAT`. The format relates to which year and whether data or MC is being used. The subformat is mainly for eras in data in which certain elements might not exist. For example, certain triggers are missing for 2017 era B, so we need to include `SUBFORMAT=DATA_2017B`. If we want to compile for MC or data in 2018, we would have the following:

```
make FORMAT=MC_2018 SUBFORMAT=MC_2018 SAMPLE=MC_NOTVZ \
PROCESSING=NORMAL NANOAOD=NANOAODV9 INPUT=TCHAIN

make FORMAT=DATA_2018 SUBFORMAT=DATA_2018B SAMPLE=DATA_NOTVZ \
PROCESSING=NORMAL NANOAOD=NANOAODV9 INPUT=TCHAIN
```

You can ignore most of the rest of the elements in these compile statements for now.

5.2 Running the Code

Once the code is compiled, it can be run using the main shell made by the makefile. The general format is as follows:

```
./make -filelist [Filelist] -out [output_area] -data [0 = no, 1 = yes] \
-year [Year] -syst [Systematic]-xccEffFileName NONE \
-centralGenWeight 0 -lastentry [N]
```

The filelists come from the `FileLists` folder and the output area is wherever you want final results to be stored. For data files, use `-data 1` and for MC, use `-data 0` . Year is either 2016, 2016PRE, 2017, or 2018. `-syst` is set as NONE for the default analysis. The names of systematics can be checked in the code (Ana.cxx, etc.) to see how we name other systematics. If `lastentry` is not included, then the code will run over all events in the file. Otherwise, it will run over N events. Technically, you can also include `-firstentry` to start from a particular event. For example, if we want to run over 2018 data era B for 10,000 events, we run the following:

```
./main -filelist FileLists_test/JetHT_DATA_2018B.txt \
-out Tmp/output_testNONE_DATA_2018B.txt.root -data 1 \
-year 2018 -syst NONE -xccEffFileName NONE -centralGenWeight 0 \
-lastentry 10000
```

5.3 Local Test Runs

There are files included that can help make sure the code runs for all eras we expect. To do so, you can run the following scripts in the following manner:

```
chmod +x test_MC_all.sh
./test_MC_all.sh
chmod +x test_data_all.sh
./test_data_all.sh
```

This runs over each year of Run 2 to help check for each year of interest.

5.4 Submitting to Condor

Rather than run everything slowly on a local machine, it is more efficient to submit everything to the computing grid to run in parallelized jobs. (This is why we build from the EDMAnalyzer base, because the code is already formatted easily in a way to be able to submit jobs.) It is a good idea to run the tests locally before submitting. Otherwise, jobs will fail. The main files in the `SubmitToCondor` files that we care about are `launch_v3.py` and `launch_v3_syst.py`. The latter just does what the former does but for multiple systematics, so we focus on the first one, `launch_v3.py`.

5.4.1 Local Paths

There are a few different paths in the script that must be defined.

sourceDir

`sourceDir` defines where our code is coming from. This is just the folder one level above `SubmitToCondor`, i.e. your workspace. In my case, I have:

```
sourceDir = '/uscms_data/d3/peteryou/boosted_new/ \
    CMSSW_14_0_6/src/VHccAnalysis/'
```

condorRunDir

`condorRunDir` is where the actual `.sh` scripts and error/log files for each sample are stored. It's useful to have a dedicated area for this. In my case, I have a special space in `/uscmst1b_scratch/lpc1/lpcphys/`, but you need to be given a space in here to use it.

outputDir_eos

You need somewhere where the output files from the jobs are placed. This will take up a lot of space since the code take each sample and break them down into smaller jobs. For a single sample, you might have 50 individual jobs that will be compiled back together. `outputDir_eos` is where these files should be stored, typically in your EOS space.

outputDir_scratch

`outputDir_scratch` is where you want the final re-compiled results to end up. This is typically a temporary folder in your workspace. Make sure this folder is included in the `.gitignore` file for the Github repository. These files will end up being GB in size and take up too much space.

5.4.2 Parameters & Settings

There are many settings to be modified when submitting to Condor. The following are relatively simple:

- `runMode` - this is set for either sending the jobs to the grid or pulling

completed jobs back to your local area. Set to 0 for submitting jobs, and 1 for retrieving completed ones.

- `submit` - this is a boolean that can be used for testing the setup. If you want to make sure the code works without actually doing the step of sending jobs to the grid, set it to `False`. Otherwise, leave it as `True`.
- `debug` - this is used for testing and forces to only submit for a small number of events (right now, it's set at 100,000 events)
- `haddData` - data must be submitted by era (A,B,C,D, etc.) but ultimately, you only want one data file per year. This helps re-combine the eras at the end. NOTE: This is currently broken!

dataSet_list

`dataSet_list` stores the dataset list you want to use for the submitted jobs. This is where you set that you only want to run over all MC, all data, a specific year, etc. What you set to run depends on what file you created in `Dataset_lists`.

dir_file_list

`dir_file_list` is where you want the files to come from. In general, we just use the `FileLists` folder which contains all the proper paths for xrootd. However, to avoid issues with data, we have a special folder `FileLists_JetHT` for those samples.

Chapter 6

Offline Analysis - Using Macros & HiggsCombine

6.1 Macros

It is useful to understand what macros we have and what they are tasked to do in this analysis. This will just go through different ones we have for different purposes.

6.1.1 Data-Driven QCD Estimation in Top CR

This method is discussed in the analysis notes. To actually go about doing it, we need to be able to run the code and then plots the results. The calculations are done using `calc_bckg_est_ratio.py`. For each year, this takes the events in the QCD-enriched top CR and calculates the transfer function with desired regions. In our case, we want to compare the top CR and the QCD-enriched top CR. We compare the number of events in QCD MC files to get the transfer function.

The only modifications to be made here are the input and output folders. The input folder should be wherever you stored your results from condor.

The output folder is wherever in your workspace that you want. There are two output files created by this script:

- `QCD_CR_evt_per_year.json` - this keeps track of how many events were seen for each sample and category that was checked for calculating the TF
- `QCD_TF_per_year.json` - this stores the calculated transfer function values for each year of Run 2. Each year only has a constant as a transfer function. **This is the most useful file for this!**

This script is run by:

```
python3 calc_bckg_est_ratio.py
```

To plot the results, run:

```
python3 plot_bckg_est.py
```

In this plotting script, you need to make sure that `json_folder` references the folder where you stored the JSON files created by the previous script. This script also has an option for blinding a given region.

6.1.2 Control Plots

To produce the control plots necessary for the analysis, you run `plot_dataMC.py`. This script sets up to produce plots for any region of interest while making sure to blind the data in the signal region (or any regions if you truly desire). To run it, you run:

```
python3 plot_dataMC.py
```

All you have to modify here is make sure the proper input files are use and that you set the regions you want.

6.1.3 Pileup Reweighting

The pileup reweighting is done in a similar way to the data-driven QCD background, in the sense that one script calculates a reweighting and the other plots. To plot the pileup distributions (from data given by Lumi POG and MC) and the consistency check, run:

```
python3 calc_PU_reweight.py
```

To produce the NPV plots before and after corrections, run:

```
python3 plot_NPVs.py
```

6.1.4 Systematics

If you want to produce a plot overlay that shows the effects of a certain systematic and its uncertainties, run:

```
python3 make_syst_unc_multiPlot.py
```

To produce the plots for all the systematics for all the samples, you need to produce a LOT of plots. This is why I created `plot_syst_unc.py` which produces the plots in bunches that you set. You have to set the following:

- `dirpath` - the input files
- `output_dir` - the output folder
- `batch_to_plot` - which set of plots to produce. The options are listed inside the macro.

This is done because the system overloads if you try to do all the samples at once. For theoretical uncertainties, run:

```
python3 theory_unc.py
```

and make sure to set the similar settings inside the macro.

6.1.5 Other Files

There are many other files. Some are useful and others are just old versions of scripts that didn't get deleted/removed. Feel free to sort through them. Some of the stuff might be useful. `my_funcs.py` that contains many custom plotting scripts.

6.2 2-prong Vtag Scale Factors

As seen in our studies and presentations, we need calibation/scale factors for our 2-prong V-tagger. This methodology was taken from HIG-24-017 and we use code borrowed from them. The setup is a bit annoying, but it is as follows.

6.2.1 Setup

The first thing to do is get the proper setup described by A. Novak in the Github repository involving this code [5]. Note that it might be good to set up a new workspace for this as well. It can also fit into the `Fit` workspace since it uses the same CMSSW version. This starts by making sure to implement `rhalphalib`. The general steps are:

```
cmsrel CMSSW_14_1_0_pre4
cd CMSSW_14_1_0_pre4/src
cmsenv
scram-venv
cmsenv
```

```
git clone https://github.com/cms-analysis/HiggsAnalysis-CombinedLimit.git \
    HiggsAnalysis/CombinedLimit
cd HiggsAnalysis/CombinedLimit
git checkout v10.1.0
scram b -j4
python3 -m pip install git+https://github.com/nsmith-/rhalphalib.git
```

There are tests given in the repository for `rhalphalib` [6] that you can run to make sure it installed properly. Once this is done, the code for the 2-prong tagger SF can be installed. So far, your working directory should look like this:

```
[your_working_space]/CMSSW_14_1_0_pre4/src/HiggsAnalysis/CombinedLimit
```

While inside this folder, do the following:

```
git clone https://github.com/andrzejnovak/TnPSF.git
cd TnPSF
```

Then, you can run the commands given in the tagger library's README file [5]. You need to modify it to use the proper files you want.

6.2.2 Additional Useful Files

At the current time where the `ttbar_selection` branch has not been merged back into the main branch, the following macros only exist on this branch, so keep this in mind when looking for them.

plot_WTag_CR

`plot_WTag_CR.py` is a script created to plot the control region used for this 2-prong V-tag stuff. This produces the plots seen on Slide 15 of our May 7th presentation [7]. Running the scripts above produces the plots shown on Slide 16.

prepare_templates

`prepare_templates.py` and `prepare_templates_v2.py` (probably just need the second one) are scripts created to produce the necessary templates that are needed for the 2-prong tagger SF code. Rather than using the templates they give, replace them with ones generated by this script.

6.3 HiggsCombine

To run the fits and produce the impact plots, we have scripts modified from HIG-24-017 that simplify things greatly for us. Make sure you are in your `Fit` workspace. There is a special repository/library that is needed for this:

```
cd Fit
git clone https://github.com/andrzejnovak/combine_postfits.git
```

Once the repository is cloned, you need to set up the environment to use it:

```
cd combine_postfits
curl https://pyenv.run | bash
export PATH="$HOME/.pyenv/bin:$PATH"
eval "$(pyenv init --path)"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
exec "$SHELL"
pyenv local 3.11.8
pip3 install -e .
```

Once the scripts are set up, you should be able to run the following scripts:

```
python3 prepareHist.py
python3 prepareFit.py
```

```
python3 runLimit.py
python3 runImpactFitDiag.py
chmod +x makePrePostFitPlot.sh
./makePrePostFitPlot.sh
```

Make sure for `prepareHist.py` to go into the script and modify the channel (around Line 163). We need to run it for both ZccHcc and VHcc before moving onto running `prepareFit.py`. To modify the plots for blinding the data in the signal region, there are modifications around Line 378 in `prepareFit.py` that can be uncommented/swapped out to do this. To have the final plots use data and not MC, go into `makePrePostFitPlot.sh` and uncomment the line about data (which just switches the flag from `--MC` to `--data`).

Bibliography

[1] Workbook 5.1 chapter overview - getting started. URL "https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookStartingGrid".

[2] D. Nugyen, P. Young. Github - vhccanalysis. URL "https://github.com/peteryouBuffalo/VHccAnalysis".

[3] 4.1.2 writing your own edanalyzer. URL "https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookWriteFrameworkModule".

[4] Nanoaod auto-documentation. URL "https://cms-nanoaod-integration.web.cern.ch/autoDoc/".

[5] Andrzej Novak. W tagandprobe rhalphalib implementation. URL "https://github.com/andrzejnovak/TnpSF".

[6] Nicholas Smith. rhalphalib. URL "https://github.com/nsmith-/rhalphalib".

[7] P. Young D. Nguyen. An-24-199 twiki. URL "https://twiki.cern.ch/twiki/bin/viewauth/CMS/VqqHcc".