Introduction to the Terascale 2014

CMS data analysis tutorial: Documentation and explanations

Jun.-Prof. Dr. Christian Sander, Dr. Alexander Schmidt

March 2014

1 Introduction

Prerequisites for this tutorial are a basic understanding of what the LHC and particle detectors are, some knowledge about top quark physics and the ROOT data analysis framework. You can follow the exercise sheet step by step which guides you to the discovery of the top quark, the measurement of the top quark production cross-section and the top quark mass.

A physics analysis of a collider experiment has several general components. The final state decay products of the studied channel are measured by the detector, so we need to develop techniques to assign the final state objects to the hypothetical signal decay cascade. For our tutorial we will restrict to semi-leptonic decay cascades of pair-produced top quarks. This is illustrated in Figure 1 which shows such a semi-leptonic $t\bar{t}$ decay tree. One W boson decays to two quarks (measured as "jets" in the detector), and the other one decays to a lepton (muon, electron, tau) and a neutrino. In addition, two b-quarks emerge from the top decay.

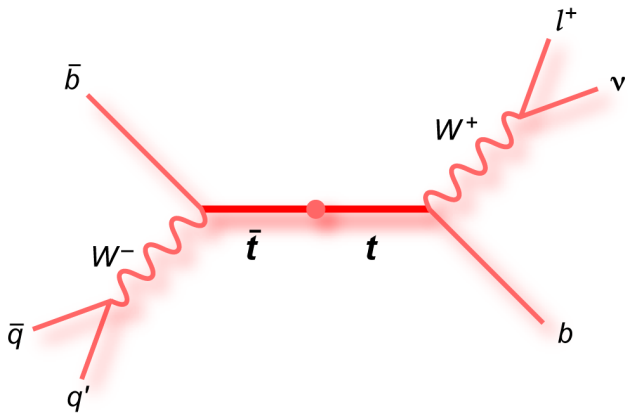

Figure 1: Feynman diagram of a semi-leptonic $t\bar{t}$ decay. The final state consists of two quarks (jets), two b-quarks (b-jets), one charged lepton and one neutrino.

The task of reconstructing the decay tree is further complicated by the presence of additional objects, such as jets from radiation or from simultaneous proton-proton collisions (pileup). Furthermore, there are plenty of other physics processes which have similar final states as $t\bar{t}$ production, for instance QCD multi-jet production or electroweak W+jets production. These processes are called backgrounds. Suppression of these backgrounds is a very important component of each physics analysis. We have to study kinematical properties of the events, such as number of jets, the jet momentum and b-tagging discriminators and others. Cuts on these variables can be determined and optimized with the help of Monte Carlo simulation.

Finally, to estimate a cross-section, we need to understand what fraction of signal events are actually selected by our analysis (so called efficiency). This efficiency includes the trigger, the detector acceptance, detector reconstruction efficiency and selection cuts on analysis level. There are sophisticated methods to measure these efficiencies precisely in a data-driven way. For the purpose of our tutorial, we can take these efficiencies from the simulation.

2 Data samples and Monte Carlo simulation

Due to the extremely high LHC collision rate of up to 20 MHz not all events can be stored. A trigger system selects the "interesting" events and reduces the total event rate to a few hundred Hertz. The trigger which has been used for this analysis selects events containing isolated single muons with a transverse momentum threshold of $p_T > 24$ GeV. The sample with real CMS data collected with this trigger has 469384 events and corresponds to an integrated luminosity of 50 pb^{-1}. Details are given in Table 1.

This table also contains information about the simulated samples. The number of simulated events can be much smaller or much larger than required for an integrated luminosity of 50 pb^{-1}, so these have to be weighted accordingly. The weight is stored in the `EventWeight` variable in each sample (more details on the variables are given in Section 3). The corresponding integrated luminosity for each MC sample, taking into account the `EventWeight` is indicated in the fifth column in Table 1 (all events have to be multiplied with their event weight to correspond to the given luminosity).

The last column of Table 1 indicates whether the sample contains also events which did not pass the trigger selection. Naturally, real data contains triggered events only. For simulation, only the $t\bar{t}$ sample also contains non-triggered events so that we can estimate the trigger efficiency for $t\bar{t}$ events.

filename	type	#events	x-section	int. lumi.	trig. only
`data.root`	data	469384		50 pb^{-1}	yes
`ttbar.root`	sim. $t\bar{t}$ signal	36941	165 pb	50 pb^{-1}	no
`wjets.root`	sim. W plus jets background	109737	31300 pb	50 pb^{-1}	yes
`dy.root`	sim. Drell-Yan background	77729	15800 pb	50 pb^{-1}	yes
`ww.root`	sim. WW background	4580	43 pb	50 pb^{-1}	yes
`wz.root`	sim. WZ background	3367	18 pb	50 pb^{-1}	yes
`zz.root`	sim. ZZ background	2421	6 pb	50 pb^{-1}	yes
`single_top.root`	sim. single top background	5684	85 pb	50 pb^{-1}	yes
`qcd.root`	sim. QCD multijet backgr.	142	10^8 pb	50 pb^{-1}	yes

Table 1: Data and simulated Monte Carlo samples.

As you see, the QCD sample contains very few triggered events, even though its production cross-section is very large. This is due to the isolated lepton requirement which strongly reduces this background already at the trigger level.

The files are contained in the package `HEPTutorial.tar.gz` in the subdirectory `files`.

3 Data format and structure of the ROOT tree

The data samples for this tutorial are stored in ROOT trees. The tree contains a collection of variables (called branches) which are filled once per event. The list of variables along with their data type and further explanations are given in the following. Variables with brackets `[]` indicate that they are stored as arrays:

- `NJet` (integer): number of jets in the event.
- `Jet_Px[NJet]` (float): x-component of jet momentum. This is an array of size `NJet`, where a maximum of twenty jets are stored (`NJet` < 21). If there are more than twenty jets in the event, only the twenty most energetic are stored. Only jets with $p_{\mathrm{T}} > 30$ GeV are stored.
- `Jet_Py[NJet]` (float): y-component of jet momentum, otherwise same as `Jet_Px[NJet]`.
- `Jet_Pz[NJet]` (float): z-component of jet momentum, otherwise same as `Jet_Px[NJet]`.
- `Jet_E[NJet]` (float): energy of the jet, otherwise same as `Jet_Px[NJet]`. Note that the four compoments `Jet_Px, Jet_Py, Jet_Pz` and `Jet_E` constitute a fourvector which fully describes the kinematics of a jet.

- `Jet_btag[NJet]` (float): b-tagging discriminator. This quantity is obtained from an algorithm that identifies B-hadron decays within a jet. It is correlated with the lifetime of the B-hadron. Higher values indicate a higher probability that the jet originates from a b-quark. **Important:** The discriminator has small performance differences in data and simulation. To account for this, simulated events have to be reweighted by a factor of ~ 0.9 per required b-tagged quark.
- `Jet_ID[NJet]` (bool): Jet quality identifier to distiguish between real jets (induced by hadronic interactions) and detector noise. A good jet has `true` as value.
- `NMuon` (integer): number of muons in the event.
- `Muon_Px[NMuon]` (float): x-component of muon momentum. This is an array of size `NMuon`, where a maximum of five muons are stored (`NMuon` < 5). If there are more than five muons in the event, only the five most energetic are stored.
- `Muon_Py[NMuon]` (float): y-component of muon momentum, otherwise same as `Muon_Px[NMuon]`.
- `Muon_Pz[NMuon]` (float): z-component of muon momentum, otherwise same as `Muon_Px[NMuon]`.
- `Muon_E[NMuon]` (float): energy of the muon, otherwise same as `Muon_Px[NMuon]`. Note that the four compoments `Muon_Px, Muon_Py, Muon_Pz` and `Muon_E` constitute a fourvector which fully describes the kinematics of a muon.
- `Muon_Charge[NMuon]` (integer): charge of the muon. It is determined from the curvature in the magnetic field and has values +1 or -1.
- `Muon_Iso[NMuon]` (float): muon isolation. This variable is a measure for the amount of detector activity around that muon. Muons within jets are accompanied by close-by tracks and deposits in the calorimeters, leading to a large values of `Muon_Iso`. On the other hand, muons from W bosons are isolated and have small values of `Muon_Iso`.
- `NElectron` (integer): same as for muons above, but for electrons.
- `Electron_Px[NElectron]` (float): same as for muons above, but for electrons.
- `Electron_Py[NElectron]` (float): same as for muons above, but for electrons.
- `Electron_Pz[NElectron]` (float): same as for muons above, but for electrons.
- `Electron_E[NElectron]` (float): same as for muons above, but for electrons.
- `Electron_Charge[NElectron]` (integer): same as for muons above, but for electrons.
- `Electron_Iso[NElectron]` (float): same as for muons above, but for electrons.
- `NPhoton` (integer): same as for muons above, but for photons.
- `Phtoton_Px[NPhoton]` (float): same as for muons above, but for photons.
- `Photon_Py[NPhoton]` (float): same as for muons above, but for photons.
- `Photon_Pz[NPhoton]` (float): same as for muons above, but for photons.
- `Photon_E[NPhoton]` (float): same as for muons above, but for photons.
- `Photon_Iso[NPhoton]` (float): same as for muons above, but for photons.
- `MET_px` (float): x-component of the missing energy. Due to the hermetic coverage of the LHC detectors and the negligible transverse boost of the initial state, the transverse momentum sum of all detector objects (jets, muons, etc...) must be zero. This is required by energy and momentum conservation. Objects which escape the detector, such as neutrinos, are causing a "missing" transverse energy which can be measured and associated to the neutrino.

- `MET_py` (float): y-component of the missing energy.
- `NPrimaryVertices` (integer): the number of proton-proton interaction vertices. Due to the high LHC luminosity several protons within one bunch crossing can collide. This is usually referred to as "pileup". The spread of these vertices is several centimeters in longitudinal direction and only micrometers in the transverse direction.
- `triggerIsoMu24` (bool): the trigger bit. It is "true" if the event is triggerd and "false" if the event is not triggered (data can only contain triggered events).

All quantities discussed above are actually measured by the detector. They are available both in Monte Carlo simulation and real data. The following variables are ONLY available for the $t\bar{t}$ signal Monte Carlo samples (described in Section 2), because they refer to the true $t\bar{t}$ decay cascade on generator level. This information can be used to study methods how to associate the simulated final state detector objects to the $t\bar{t}$ decay cascade which can eventually be applied also to real data. Please note that these variables are only filled for semi-leptonic $t\bar{t}$ decays. They are set to zero for fully hadronic and fully leptonic decays. They are also zero in the background Monte Carlo samples.

- `MChadronicBottom_px` (float): x-compoment of the b-quark from the top decay belonging to the hadronic branch.
- `MChadronicBottom_py` (float): y-compoment ...
- `MChadronicBottom_pz` (float): z-compoment ...
- `MChadronicWDecayQuark_px` (float): x-component of the quark from the hadronic W boson decay
- `MChadronicWDecayQuark_py` (float): y-component ...
- `MChadronicWDecayQuark_pz` (float): z-component ...
- `MChadronicWDecayQuarkBar_px` (float): x-component of the anti-quark from the hadronic W boson decay
- `MChadronicWDecayQuarkBar_py` (float): y-component ...
- `MChadronicWDecayQuarkBar_pz` (float): z-component ...
- `MCleptonicBottom_px` (float): x-compoment of the b-quark from the top decay belonging to the leptonic branch.
- `MCleptonicBottom_py` (float): y-compoment ...
- `MCleptonicBottom_pz` (float): z-compoment ...
- `MClepton_px` (float): x-component of the lepton (electron, muon, tau) from the leptonic W boson decay.
- `MClepton_py` (float): y-component ...
- `MClepton_pz` (float): z-component ...
- `MCleptonPDGid` (integer): particle "ID" of the lepton. Possible values are 11 for electrons, 13 for muons, 15 for taus. Negative numbers indicate anti-particles.
- `MCneutrino_px` (float): x-component of the neutrino from the leptonic W boson decay.
- `MCneutrino_py` (float): y-component ...
- `MCneutrino_pz` (float): z-component ...
- `EventWeight` (float): weight factor to be applied to simulated events due to different sample sizes.

4 Analysis framework

The package `HEPTutorial.tar.gz` contains an example framework to help you getting started. Unpack the example with `tar xvzf HEPTutorial.tar.gz`. You can build your additional analysis code on top of this example. The example is already running, but it's not doing much yet. You can compile the analysis code by first initializing a ROOT environment followed by executing the command `make`.

This will read the necessary ingredients for compilation from the `Makefile` in the same directory. You don't need to understand the `Makefile` at this stage. The important point is that it creates an executable named `example.x`. You can then simply execute the program by the command `example.x`.

A description of the individual components of the example are given in the following list. Indicated are also the places where you should start adding your own code:

- `example.C`: this is the first starting point. It contains the `main()` function which is necessary for any C++ program. The first step is to create instances of `MyAnalysis` which is implemented in the files `MyAnalysis.h` and `MyAnalysis.C` (explained in the next item). The `TChain` represents the ROOT tree discussed in Section 3. The files which should be read from disk are specified in the function `Add(filename)`. The tree is then read and processed by the `Process()` function which takes a `MyAnalyis` as argument. The real work is then done in the `Process()` function of the `MyAnalysis` class which is discussed in the next two items. You should instantiate a separate `MyAnalysis` class for each data or MC sample that you want to process. The end of the `main()` function shows how to access and write out the histograms which are filled inside the `MyAnalysis` class.

- `MyAnalysis.h`: definition of the class `MyAnalysis`, which inherits from a ROOT `TSelector` (see the ROOT reference manual for details about `TSelector`). The constructor takes a global scaling factor (by default "1.0") which is applied to all events in the given sample (multiplied to the individual event weights). This global scaling factor can be used to normalize the MC cross-sections. `MyAnalysis` contains the declaration of all variables (such as `Jet_Px`) which are contained in the ROOT tree (see section 3). It also declares variables and functions that will be used in your analysis, such as histogram pointers (type `TH1F*`). Some containers of type `vector<...>` are also declared here. These containers will hold the helper classes representing jets (`MyJet`) or muons (`MyMuon`). Finally, the `MyAnalysis::Init()` function makes the connection between the ROOT tree (stored on disk) and the variables which are kept in memory.

- `MyAnalysis.C`: the two main functions which are called automatically while processing the ROOT trees are `SlaveBegin()` and `Process()`. The `SlaveBegin()` function is called only once per job, just before the processing of the events start. You see that it is used for booking histograms (assigning the histograms to the pointers defined in the MyAnalysis.h file). The `Process()` function is called automatically for every single event. This is the place where the core of the analysis happens. In the existing example `Process()` calls a subroutine `BuildEvent()` which takes care of filling some of the kinematic variables into convenient representations as `MyJet` or `MyMuon`, which are essentially fourvectors (explained in the next item). After building the event, a very simple example analysis is performed in `Process()` which fills the transverse momentum p_t of a muon into a histogram.

- `MyMuon.h`/`MyMuon.C`: inherits from `TLorentzVector` which is basically a fourvector (see the ROOT reference guide for details about `TLorentzVector`. It adds additional information about the muon charge and muon isolation variables to the fourvector.

- `MyElectron.h`/`MyElectron.C`: same as `MyMuon` but for electrons.

- `MyJet.h`/`MyJet.C`: inherits from `TLorentzVector`, adds additional information about the b-tagging variable to the fourvector. The b-tagging variable is correlated to the probability that the jet originates from a b-quark. Large values of this variable means high probability for a b-quark jet. The function `IsBTagged()` applies a cut to the b-tag discriminator and returns a boolean decision (true or false). The function `GetJetID()` returns `true` if a jet fulfills basic quality criteria, else it returns `false`.

- `Plotter.h`/`Plotter.C`: A tool which can be used for automatic plotting of a set of histograms which are stored in a `std::vector`. Please see `example.C` on how to use it.

The program writes an output file , e.g. `results.pdf`, which contains the plots from the analysis. You can open the file with any `pdf` viewer, for example `acroread`.