

# TopRun2WorkshopPreTutorial2020

Not yet  
Certified as  
ATLAS  
Documentation

**Responsible:**  
(no spam please!)

[MarcoVanadia](#)

- ↓ [1 Prerequisites](#)
- ↓ [2 Setting up and playing with AnalysisTop](#)
  - ↓ [2.1 Make a flat ntuple containing only corrected and selected objects](#)
  - ↓ [2.2 Modify the event selection locally](#)
  - ↓ [2.3 Compare the two output files](#)
  - ↓ [2.4 Run the selection together with event reconstruction](#)
  - ↓ [2.5 More modifications of the configuration file](#)
    - ↓ [2.5.1 More modifications of the configuration options](#)
    - ↓ [2.5.2 Systematic modifications](#)
    - ↓ [2.5.3 More modifications of the event-level selections](#)
    - ↓ [2.5.4 Using particle-level selections](#)
  - ↓ [2.6 Browsing the AnalysisTop code](#)
- ↓ [3 Extending AnalysisTop](#)
  - ↓ [3.1 Setting up the HowToExtendAnalysisTop package](#)
  - ↓ [3.2 Creating a custom-saver](#)
    - ↓ [3.2.1 The very basic](#)
    - ↓ [3.2.2 Let's add something real to the output now](#)
    - ↓ [3.2.3 Getting access to the config options and retrieving containers](#)
    - ↓ [3.2.4 Using dynamic keys](#)
    - ↓ [3.2.5 Adding some particle-level stuff](#)
  - ↓ [3.3 Creating a Custom Selector](#)
- ↓ [4 Launching jobs on the grid](#)
  - ↓ [4.1 Retrieving the TopExamples grid scripts](#)
  - ↓ [4.2 Launching jobs on the grid](#)
  - ↓ [4.3 baby-sitting the jobs](#)
  - ↓ [4.4 I have a sample to run on but there is no TOPQX derivation available](#)
  - ↓ [4.5 other useful features](#)
  - ↓ [4.6 Taming grid issues](#)
  - ↓ [4.7 Monitoring dataset obsolescence](#)
- ↓ [5 Normalisation to the integrated luminosity](#)
  - ↓ [5.1 Retrieving the integrated luminosity](#)
  - ↓ [5.2 Retrieving the number of processed MC events](#)
  - ↓ [5.3 Retrieving the cross-section of the MC sample](#)
- ↓ [6 Internal generator reweighting](#)
- ↓ [7 Some more details on PRW](#)
  - ↓ [7.1 Inspecting the PRW files](#)
  - ↓ [7.2 Generating my own PRW file](#)
  - ↓ [7.3 Using actual mu reweighting](#)
- ↓ [8 More on b-tagging and CP recommendations](#)
- ↓ [9 The TopDataPreparation file: cross-sections and showering algorithms](#)
- ↓ [10 Saving disk-space](#)
  - ↓ [10.1 Using DynamicKeys to control my CustomEventSaver](#)
  - ↓ [10.2 Filtering out branches and/or trees](#)
  - ↓ [10.3 Do not just dump particle/parton level stuff into the reco trees](#)
- ↓ [11 How to report a problem or request a new feature](#)
- ↓ [12 Running multiple configuration files at the same time](#)
  - ↓ [12.1 First let's do a test locally](#)
  - ↓ [12.2 Merging output from different config files](#)
  - ↓ [12.3 Analysing the combined output: synchronizing different TTrees](#)
  - ↓ [12.4 Doing this on the GRID](#)
- ↓ [13 What if I need to change the AnalysisTop code?](#)

## 1 Prerequisites

Here are the prerequisites for this tutorial. You should already have received these instructions already, but here they are just in case.

You need to make sure you have an ATLAS account, with sufficient space (2 GB) in your work directory.

- If you do not have an account yet, you may follow the instructions [here](#)
- To manage your lxplus account if/once you have one, you can go [here](#)
- To extend your work directory space, you can go [here](#)
- You need a valid grid certificate ready to be used on lxplus, if you don't have it follow instructions [here](#)

You need to make sure you have access to CERN GitLab; while working on lxplus, you will need to follow the instructions at [this page](#).

If you prefer to use SSH authentication (e.g. if you have issue with using Kerberos on your laptop later), you'll have to tell GitLab what your public key(s) is/are. Click on the portrait in the upper right, choose "Settings", select "SSH Keys" on the left, and follow the instructions.

If you're new to ATLAS, it's a VERY good idea to join an ATLAS induction day and a software tutorial, or to at least read very carefully the slides from the last available one if the next one is quite far; you can find the agendas [here](#).

Also, subscribe to the hn-atlas-top-reconstruction mailing list (you can subscribe using [this website](#) and make sure to follow [Top Reconstruction meetings](#)).

## 2 Setting up and playing with AnalysisTop

From now on, we'll assume you have a work directory on lxplus. It is typically located at `/afs/cern.ch/work/<l>/<login>` where `<login>` represents your login, and `<l>` its first letter (example: `/afs/cern.ch/work/t/tpelzer`). We'll assume from now on that you've done a symbolic link to `~/work` (e.g. by doing: `ln -s /afs/cern.ch/work/<l>/<login> ~/work`). Let's now setup a directory to play with AnalysisTop a bit.

AnalysisTop is now included in the AnalysisBase release. If you're not sure what we mean with "AnalysisBase release", you should probably follow an ATLAS Software tutorial as suggested above. Anyway, this is basically a very large collection of packages and executables, including the dozen or so of AnalysisTop packages. Releases have identifying numbers: we'll use here releases of the "21.2" series, which is the standard right now for run-2 analyses. The complete number is something like "21.2.XXX", with the last digits representing a specific version of the software: usually a new one is built every week. **We will use release 21.2.119 for the tutorial**, so make sure to use that in the setup below. [Release notes](#) are provided every week, with the list of changes applied in any release; we highlight the most important changes for AnalysisTop in [this twiki page](#), which presents a subset of the general release notes. It's not important to look carefully at those notes now, but bookmark the pages, you may need it for the future.

Full information on how to setup areas are on [the dedicated twiki page](#). We include here the relevant part. We'll use the method below for the setup in the tutorial, but for the future you may want to use an automatic setup for an area with a customer saver, see [this twiki](#).

First create the directory structure

```
mkdir MyAnalysis
cd MyAnalysis
mkdir build source
```

Then do the basic ATLAS setup

```
setupATLAS
lsetup git
```

Setup the analysis release you want in the source directory

```
cd source
asetup AnalysisBase,21.2.XXX,here #adjust the release number to the one you want to use, you can find here https://twiki.cern.ch/twiki/bin/view/Atlas
```

You should notice a file called CMakeLists.txt has been created in the directory. Go back to the home directory:

```
cd ..
```

Then open a file called `setup.sh` and add the following lines to that:

```
setupATLAS --quiet
lsetup git
cd source/
asetup --restore
cd ../build
source */setup.sh
cd ..
#lsetup panda
#lsetup pyami
#voms-proxy-init --voms atlas
```

(note: rucio cannot be setup together with an [AnalysisRelease](#), if you need that you'll need to work in a separate terminal/shell)

this will be used the next times you will start working in this area (uncomment the last lines if you plan to use the GRID and if you have a valid certificate for that).

Let's prepare some compilation scripts. Go to the work directory MyAnalysis ("cd .." if you're still inside source/ from the previous step).

Open a file called `full_compile.sh` and add the following lines:

```
cd build
cmake ../source
cmake --build ./
source */setup.sh
cd ..
```

and then give the command:

```
source full_compile.sh
```

You can also prepare another script called `quick_compile.sh`, with the following lines

```
cd build
cmake --build ./
source */setup.sh
cd ..
```

You will have to use the `source full_compile.sh` for the first compilation and every time you add new classes or change package dependencies; if you just modify the code of existing files, you can just do `source quick_compile.sh`.

### 2.1 Make a flat ntuple containing only corrected and selected objects

Now that you've made the setup, let's try to run over some sample. First, inside the workdir you created at the beginning, create a `run` directory and go there:

```
mkdir run
cd run/
```

Then, we just need to run the main AnalysisTop executable. This code, `top-xaod`, takes the input and applies the CP corrections and object selection. After that it does the event-level cuts specified in the configuration file, with only corrected and selected objects written out, and only for those events that pass the selection.

To jump straight in we can try to run on some events from a test MC16 ttbar sample. The code needs two inputs:

- a configuration file that specify the object definition, settings and cuts
- a file that has a list of samples to run over

So, let's take an example config file and play with it:

```
cp $AnalysisBase_DIR/data/TopAnalysis/example-config.txt my-cuts.txt #we will play with the my-cuts config file in the following
```

For the filelist, you can just run on one file of a ttbar MC sample. Prepare the filelist with:

```
\ls /cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/ContinuousIntegration/MC/p4031/mc16_13TeV.410470.PyPy8EG_A14_ttbar_hdamp258p75_nor
```

Now, open `my-cuts.txt` with your favourite text editor, find the line containing:

```
#NEvents 500
```

This configuration option sets how many events you want to run on. The `#` in front means that that option is commented, i.e. the default setting (running on all events in the input files) is used. Let's run on 5000 events for a first test; change the line in the config file to:

```
NEvents 5000
```

This will make you run over 5000 events only, while the input file contains 60000 events in total (and we don't have much time). It will take a while (and we're running just the nominal setup, without no systematic variations, for now). Then, still from your run directory, do:

```
top-xaod my-cuts.txt input-mc-rel21_p4031.txt
```

You will likely get warnings like

```
top::GhostTrackSystema...WARNING in GhostTrackSystematicsMaker: All ghost tracks are null pointers. There may be something wrong with your configuratic
and
```

```
ackage.MuonEfficiency WARNING The dRJet decoration has not been found for the Muon. Isolation scale-factors are now also binned in #Delta R(jet,#mu)
Package.MuonEfficiency WARNING using the closest calibrated AntiKt4EMTopo jet with p_{T}>20~GeV and surviving the standard OR criteria.
Package.MuonEfficiency WARNING You should decorate your muon appropriately before passing to the tool, and use dRJet = -1 in case there is no jet in a
Package.MuonEfficiency WARNING For the time being the inclusive scale-factor is going to be returned.
Package.MuonEfficiency WARNING In future derivations, muons will also be decorated centrally with dRJet, for your benefit.
```

these can be ignored.

**At the beginning of the execution you will see that top-xaod prints out ALL the AnalysisTop options that are used. The ones you did not set in the config files take the default values.** You should see something like below:

We will familiarize with some of the most important options in this tutorial; you can find also important information on the most relevant options in the [start guide](#) (after completing this tutorial!).

Now the event selection is running and once its done you should see in your shell many cutflows. Each cutflow corresponds to one event selection encoded in the configuration file. One of the last things printed is a message like:

```
Events saved to output file nominal reconstruction tree: 590
```

This is showing how many events are stored in the output `nominal` tree: these are the events passing **at least one** of the event-level selection defined in your config file. We'll discuss selections later.

If you now do an `ls` in your folder, you will see that you have now a file called `output.root`. Please open it:

```
root -l output.root
```

You can have a look at the content with

```
[ ] .ls
```

and also browse with a TBrowser:

```
[ ] new TBrowser()
```

You will see several folders and five trees inside. Lets start with the folders: one folder for each **event-level selection** is available. If you click for example on the folder called `ejets_DL1r_2016` (which is one of the event-level selections defined in the config file), you will find the content of the cutflow stored in histograms: double-click on "cutflow" to plot it. Furthermore you find histograms for the events that survived the electron+jets selection designed for 2016 data, containing information about all objects and event information like weights, missing ET, etc.

Close the folder in the TBrowser (with the small minus sign) and let's now have a look at the trees. The most important tree for today is the `nominal` tree: double-click on it to open it in the browse: sure we have a lot of branches! It contains the information of all five selections together stored in branches and allows for further analysis of the event. For each type of selection there is a boolean variable stored that tells you if the event passed the specific selection or not. For example, double-click on the branch `ejets_DL1r_2016`: you will see a plot with one entry for each event stored in the nominal tree, with a "0" for events NOT passing the

ejets\_DL1r\_2016 selection, and a "1" for events passing that selection. You can also check that from the command line:

```
[ ] nominal->GetEntries()  
[ ] nominal->GetEntries("ejets_DL1r_2015")  
[ ] nominal->GetEntries("ejets_DL1r_2016")  
[ ] nominal->GetEntries("ejets_DL1r_2017")  
[ ] nominal->GetEntries("ejets_DL1r_2018")
```

Hey, why are the selections for 2017 and 2018 empty? Well, we're using a file from the sample

mc16\_13TeV.410470.Py8EG\_A14\_ttbar\_hdamp258p75\_nonallhad.deriv.DA0D\_T0PQ1.e6337\_s3126\_r9364\_p4031 . From the `r`-tag, which in this case is `r9364`, one can understand that this is a [mc16a](#) sample, i.e. a sample meant to simulate data-conditions for 2015/16. For 2017, you would have to use the version of this sample with `r10201`, corresponding to the [mc16d](#) campaign. For 2018 you have `r10724`, corresponding to the [mc16e](#) campaign. So, when you'll run a full run-2 analysis, you will have to run on the 3 version for each of your samples and combine them. But we'll not do this for the tutorial, just keep this in mind for the future.

The `nominal` tree is filled with events passing selections defined on reconstructed objects. The other trees are `truth` (filled with truth information, for all process events on MC), `particleLevel` (filled with truth information for events passing selections defined on particle-level objects; you are not seeing it now because it's turned off in the config file), `sumWeights` (filled with some useful meta-data, once for each processed input file), and `AnalysisTracking` (which is a new feature in recent releases, created for the purpose of analysis tracking).

Try to get familiar with the content of the `nominal` tree, looking at the variables inside, drawing some of them, etc... . Some examples: open the file in root `root -l output.root` and then do things like:

```
nominal->Print() #this will print the names of all branches of the tree  
nominal->Print("weigh*") # this will print all branch names starting with "weigh"  
nominal->Scan("el_charge:el_pt","el_pt[0]>30000.") #this will print the values of electron charge and pt for electrons in events with the leading elec  
nominal->Draw("mu_pt[0]/1000. >> histo(20,0,100)","mu_charge[0]>0") #this will draw an histogram of the leading muon pt (in GeV, hence the /1000.) f  
nominal->Draw("mu_pt[0]/1000. >> histo(20,0,100)","((mujets_DL1r_2015>0 || mujets_DL1r_2016) && met_met>50000.)*weight_mc*weight_pileup*weight_lepton"
```

Of course we ran on relatively few events, so the plots will not be super-nice, but ok, you should be able to get an idea... feel free to re-run increasing the number of events you use if you have time!

It's particularly important to familiarize with the weights available in the tree, which usually must be used for MC samples. You will for example find:

- `weight_mc`: MC events have a generator level weight that must be applied when they are used (unless you know what you're doing...).
- `weight_pileup`: to reweight the pileup distribution used to generate the MC sample to the one expected in data. More info [here](#).
- `weight_leptonSF` (and `photonSF`, `tauSF`): SFs that reweight various kind of efficiencies (reconstruction, identification, isolation...) in MC to reproduce the data ones.
- `weight_globalLeptonTriggerSF`: global trigger SF for leptons (ignore `weight_oldTriggerSF` unless you're using an older approach, and you usually shouldn't)
- `weight_bTagSF_MV2c10_77`: and similar: b-tagging SFs, there are several types for the different taggers you configured in the config file.
- `weight_jvt`: SF to reweight the efficiency of the Jet Vertex Tagger in MC to look like the data one, see [this twiki](#).
- ...

You usually should apply the product of several of these weights as an event weight as shown above, or when filling an histogram (e.g. `histo_mupt->Fill(mu_pt/1000.,weight_mc*weight_pileup*weight_leptonSF)` ). The correct thing to do depends on your analysis. In the tree you can also see systematic variations of the weights, to calculate the impact of systematic uncertainties (for example, you can fill `histo_mupt_pileupUP->Fill(mu_pt/1000.,weight_mc*weight_pileup_UP*weight_leptonSF)` and compare it with the nominal `histo_mupt` to check the impact of the pileup UP variation on your muon pt distribution.

## 2.2 Modify the event selection locally

Now let's have a closer look at the exact settings and play with them. Open the file `my-cuts.txt` to examine its content. As a first step, please find the line:

```
OutputFilename output.root  
## and modify it to:  
OutputFilename output-modifiedSelection.root
```

In this way, we won't overwrite the first output file that was created. Now let's modify the electron+jets selection. For that, go further down in the steering file `my-cuts.txt` and find the line starting with `SUB ejets_basic`. Let's have a look at this:

```
SUB ejets_basic  
EL_N 25000 == 1  
MU_N 25000 == 0  
GTRIGMATCH  
JETCLEAN LooseBad  
JET_N 25000 >= 1  
JET_N 25000 >= 2  
JET_N 25000 >= 3  
JET_N 25000 >= 4  
MET > 30000  
MWT > 30000  
#RECO::KLFITTERRUN kElectron KLFitterBTaggingWP:DL1r:FixedCutBEff_77  
#KLFITTER > -50.0  
EXAMPLEPLOTS  
#PRINT  
NOBADMUON
```

`SUB ejets_basic` means that this is a sub-selection, i.e. a part of an event-level selection, that can be then included in final event-level selections, and we call it `ejets_basic`. If you have a look at few lines below, you will see that where you declare `ejets_DL1r_2015`, `ejets_DL1r_2016`, etc... (e.g. with `SELECTION`

ejets\_DL1r\_2015) we are including this sub-selection with the line `. ejets_basic`. This means that the cuts defined for `ejets_basic` are included in all the selections including that with `. ejets_basic`. Sub-selections are a very good way of applying the same cuts for several selections: instead of copying and pasting many lines in different selections (with the risk that then we have to modify them, and we forget to do the same change in many different places), you just use a sub-selection and include it wherever you need it.

After the definition, we're requiring to have exactly 1 electron with  $pt > 25$  GeV and exactly no muon with  $pt > 25$  GeV. Note that these **event-level** cuts are applied only on objects passing the **object-level** definitions. E.g. if you have a muon-configuration which uses the option **MuonPt 30000** you will only be able to see muons with  $pt > 30$  GeV, so it doesn't make much sense to then apply at event-level a lower cut... the same applies to jets, electrons, etc... We'll discuss this again later.

We'll discuss other cuts in more detail in the following of the tutorial. Let's focus on the lines

```
JET_N 25000 >= 1
JET_N 25000 >= 2
JET_N 25000 >= 3
JET_N 25000 >= 4
```

Here we're requiring to have at least 1 jet with  $pt > 25$  GeV, then at least 2 jets with  $pt > 25$  GeV, and so on... Of course in the end the selection will apply the tightest cut, i.e. will only store events with at least 4 jets with  $pt > 25$  GeV. Why don't we just have the last of the four lines then? Well, because in this way in the cutflow (that you saw already previously) you will have all these 4 steps, and you can check how much each one of those requests reduces the number of events you select!

Now let's modify something! Let's start with the cut on the missing transverse energy (MET) that in your previous selection a few minutes ago was defined as  $> 30000$  MeV. Change it to  $20000$  MeV and close the file. Run the selection again now:

```
top-xaod my-cuts.txt input-mc-rel21_p4031.txt
```

Now you should see in the output on the screen, that the cutflows for all channels are unchanged, except for all the `ejets` selections (ALL of the selections including `ejets_basic` are modified, if we change `ejets_basic`) for example for `ejets_DL1r_2016`.

## 2.3 Compare the two output files

You have now two output root files that contain trees as well as histograms. Open the two files separately and plot the met distribution for different selections, e.g.:

```
nominal->Draw("met_met/1000. >> histo(20,0,200)","(ejets_DL1r_2015>0 || ejets_DL1r_2016)*weight_mc*weight_pileup*weight_leptonSF")
nominal->Draw("met_met/1000. >> histo(20,0,200)","(mujets_DL1r_2015>0 || mujets_DL1r_2016)*weight_mc*weight_pileup*weight_leptonSF")
```

Are the `met` distributions for the two files different for `mujets` selections? And for `ejets` selections?

In order to do a quick comparison for many plots, we have a script provided for you that can do that. First make the output folder for the plots:

```
mkdir comparison_plots
```

Let's grab the script:

```
cp $AnalysisBase_DIR/bin/validation.py my-validation.py
```

Now open the `my-validation.py` file. Search for these two lines:

```
channels = ['ejets_2015', 'ejets_2016', 'mujets_2015', 'mujets_2016', 'ee_2015', 'ee_2016', 'mumu_2015', 'mumu_2016', 'emu_2015', 'emu_2016']
particles = ['Electrons', 'Muons', 'Photons', 'Jets', 'Tau', 'MET']
```

we only have electrons, muons, MET and jets defined in our config file, and we have different selection names. So change the lines to:

```
channels = ['ejets_DL1r_2016', 'mujets_DL1r_2016']
particles = ['Electrons', 'Muons', 'MET']
```

(we removed also 'Jets' just to avoid confusion due to the many plots which are produced, but you can keep them in if you like).

And now run the python script as follows:

```
python my-validation.py comparison_plots output.root output-modifiedSelection.root
```

If you now do an `ls` in the `comparison_plots` folder, you should be able to see a long list of plots, as well as an html file. To easily open this html file, you may need to copy locally on your laptop the `comparison_plots` directory. In a new terminal, simply do:

```
scp -r <login>@lxplus.cern.ch:~/work/MyAnalysis//run/comparison_plots/ . #or whatever the correct path to that folder is
```

where `<login>` represents your logging. You can either look directly at the images or open the `index.html` file in your favourite browser to visualize them easily. Can you find the change in your missing ET distribution for the `ejets_DL1r_2016` selection?

## 2.4 Run the selection together with event reconstruction

In many analyses it is necessary not only to select the events, but also make a full event reconstruction to be able to reconstruct the top quark or W-boson quantities. One possibility to do that is to use the kinematic likelihood fitter package `KLFitter`. The most common application of `KLFitter` is the reconstruction of top-quark pair events at the LHC, i.e. the determination of the association of measured objects, in particular jets, to the decay products of the top-quark pair decay. This package is already part of `AnalysisTop` and can be easily run with your event selection. To test this, please open the `validation-cuts.txt` file again and modify the following values:

```
OutputFilename output-modifiedSelection.root
## and modify it to:
OutputFilename output-modifiedSelection_KLFitter.root
```

and then find for the `ejets` and `mujets` selection the following lines:

```
#RECO::KLFITERRUN kElectron KLfitterBTaggingWP:DL1r:FixedCutBEff_77
[...]
#RECO::KLFITERRUN kMuon KLfitterBTaggingWP:DL1r:FixedCutBEff_77
```

and uncomment them by removing the "#". These two lines enable KLfitter for the selections where they are used; the first one is relevant for the electron+jets channel (as you can easily guess, since it's inside `ejets_basic`), and the second one for the muon+jets channel.

The event reconstruction is a bit CPU intensive, so we run it only over a subset of events. This is done by setting `NEvents` to 500:

```
NEvents 500
```

in the `my-cuts.txt` file.

Now run the selection again:

```
top-xaod my-cuts.txt input-mc-rel21_p4031.txt
```

Let us repeat: the event reconstruction takes more CPU to calculate all possible jet permutations. So only use the KLfitter in your selections if you really need it in your analysis!

Now open the output file:

```
root output-modifiedSelection_KLfitter.root
new TBrowser()
```

and you will see that the variables calculated for the KLfitter are stored in the nominal tree as well. If you want more information on the KLfitter, have a look at [twiki page](#) or at the [README](#) of the package.

## 2.5 More modifications of the configuration file

As you may have seen already, many things are specified in the configuration file. There are basically two parts: the first part is the global configuration of the run (for example you specify the name of the outfile as explained above), and the second part is the definition of the event selection(s).

### 2.5.1 More modifications of the configuration options

There are *many* options you may change in the configuration file. They should be listed at the beginning of a `top-xaod` run. You may also find them in the code ([this is the version in release 21.2.119 we are using for the tutorial](#)), and [this is the most recent \(HEAD\) version](#)). A full guide for most common options is [here](#). Let's now have a look at some of them.

```
LibraryNames libTopEventSelectionTools libTopEventReconstructionTools
```

This is to specify which libraries should be loaded. For the moment you don't have to change anything here, but you may have to add some additional libraries if you make your own extension of `AnalysisTop` (see later).

```
ObjectSelectionName top::ObjectLoaderStandardCuts
OutputFormat top::EventSaverFlatNtuple
OutputEvents SelectedEvents # AllEvents-> store all events, SelectedEvents -> store events passing at least one of the event-level selections (defined
OutputFilename output-modifiedSelection_KLfitter.root
```

You should already have modified `OutputFilename` at this stage. `OutputFormat (ObjectSelectionName)` defines the output format (object selection) - and we'll see later how to create another one. `OutputEvents` selects if you want to store in the output ntuple only events passing at least one of the event-level selections you define in the config file, or all events.

```
UseAodMetaData True
#IsAFII False
```

In the past, it was complicated to know from the input sample meta-data whether this sample is FS or AF2. Therefore we had to specify `IsAFII True` when running over AF2 MC samples. Nowadays it's not necessary anymore; you just need to tell `AnalysisTop` to trust the meta-data, by setting `UseAodMetaData True`, and if will ignore completely what you put in the `IsAFII` field. On old derivations, `UseAodMetaData True` may actually not work, unfortunately, but on the derivations you will use on your actual analysis it should be fine.

```
GRLDir GoodRunsLists
GRLFile data15_13TeV/20170619/physics_25ns_21.0.19.xml data16_13TeV/20180129/physics_25ns_21.0.19.xml
PRWConfigFiles_FS dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16a.FS.v2/prw.merged.root
PRWConfigFiles_AF dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16a.AF.v2/prw.merged.root
PRWLumiCalcFiles GoodRunsLists/data15_13TeV/20170619/PHYS_StandardGRL_All_Good_25ns_276262-284484_ofLumi-13TeV-008.root GoodRunsLists/data16_13TeV/20180129/PHYS_StandardGRL_All_Good_25ns_276262-284484_ofLumi-13TeV-008.root
```

This part sets up the good run lists (GRL) and the pileup reweighting (PRW). In the `GRLFile` field you should give one or several good run lists separated by spaces. The code will look for them in the `GRLDir` directory. As for other such kind of file path options, this directory will first be searched for [here](#), and then locally. It is in general preferable to use "official" good run lists.

The `PRWLumiCalcFiles` give the distribution of pile-up on data; the files you specify here should correspond to the good run lists you specify above. The `PRWConfigFiles` lists the PRW files which give the distribution of pile-up on MC. Pile-up reweighting has become more complex in release 21 because we have to deal with different MC sub-campaigns, each having its own pile-up distribution (mc16a corresponding to 2015/16 data conditions, mc16d corresponding to 2017 data conditions, mc16e corresponding to 2018 data conditions). As we discussed above, you can understand if you're running on a mc16a, mc16d or mc16e sample from the `r`-tag of the sample:

- r9364 -> mc16a



- r10202 -> mc16d
- r10724 -> mc16e

For the PRW you can use either a centrally provided PRW file (as we're doing here), or specify your own (which you would have generated following the instructions of the [ExtendedPileupReweighting](#) twiki). You may have to use different PRW files for fast (AF2) and full (FS) sim samples, in which case you could use the `PRWConfigFiles_AF` and `PRWConfigFiles_FS` options instead. For the mc16a campaign, the "standard" PRW is recommended, but for mc16d and mc16e actualMu reweighting is recommended. The recommended settings for AnalysisTop for all cases are kept up to date at [this page](#), have a look at that and always make sure to use up-to-date settings for your analyses!

Sometimes it may happen that some particular MC sample is not available in the central pileup files: refer to the [ExtendedPileupReweighting](#) twiki to generate a custom one, or open a JIRA ticket on the [AnalysisTop JIRA board](#) to ask for help.

```
ElectronCollectionName Electrons
MuonCollectionName Muons
JetCollectionName AntiKt4EMPFJet_BTagging201903
LargeJetCollectionName None # can be e.g. AntiKt10LCTopoTrimmedPtFrac5SmallR20Jets
TauCollectionName None #can be e.g. TauJets
PhotonCollectionName None #can be e.g. Photons
TrackJetCollectionName None #can be e.g. AntiKtVR30Rmax4Rmin02TrackJets
JetGhostTrackDecoName None # can be e.g. GhostTrack
```

These are for specifying which object collections you want to use. If for electrons, muons, taus, or photons there may not be much choice, for jets it matters - for example you can try `JetCollectionName AntiKt4EMTopoJets_BTagging201810`. If you put `None` for one of these collections, then you would ignore the corresponding object completely; however, you should make sure to not use this object in the selections afterwards.

```
### Electron configuration
ElectronPt 25000
ElectronID TightLH
ElectronIDLoose LooseAndBLayerLH
ElectronIsolation FCTight
ElectronIsolationLoose None

### Muon configuration
MuonPt 25000
MuonQuality Medium
MuonQualityLoose Medium
MuonIsolation FCTight_FixedRad
MuonIsolationLoose None
doExtraSmearing False
do2StationsHighPt False
```

These options are for electron and muon identification and isolation. The possible options should be those provided by the relevant performance group. As you can see there is for each a `Loose` option; this allows to specify a looser option for identification and isolation. The selections may be processed for both `Loose` and not `Loose` version, and you may have a `nominal_Loose` tree in addition to the `nominal` one, which is produced using the `loose` definitions you configured for leptons, instead of the standard one. The idea usually is to have a look at the loose selections in data, enriched in multi-jet background contribution - for example if you use the [matrix-method](#) for fake prompt leptons estimates. To use this feature, you may play with the following options you will find at the beginning of the example config file:

```
# DoTight /DoLoose to activate the loose and tight trees
# each should be one in: Data, MC, Both, False
DoTight Both
DoLoose Data
```

You can try to switch `DoLoose` to `Both` instead of just `Data`, and run again over the MC input file. You should see the additional `nominal_Loose` tree. Note that there are several reasons for willing to have a loose selection. For example, one may work only with the `nominal_Loose` tree in order to pre-select the objects based on the loose selection, and then apply the tight cuts afterwards (for this you may need to use the `EL_N_TIGHT` and `MU_N_TIGHT` selectors in your event-level selections).

The `ElectronID`, `ElectronIsolation`, `MuonQuality`, `MuonIsolation`, etc... options control what is the "standard definition" of what an electron/muon is for you. As usual, you can refer to the [ConfigurationSettings.cxx](#) file to check all available options: for example, you can use the `ElectronPt` option to control what is the minimum pt cut (note: everything is in always in MeV!) that an electron has to satisfy to be considered "an electron for your analysis". It's **extremely** important to understand the difference between these **object definitions** (or object-level selections) and the **event-level selections** that are then applied. For instance:

```
ElectronPt 25000.
```

is a **object definition** (or object-level selection)

while the line appearing in some of the selections at the end of the config file

```
EL_N 25000 >= 1
```

is an **event-level selection**. What is the exact difference? If you set `ElectronPt 25000`, no electron below 25 GeV will be considered at all, so even if you later will have some selection as e.g. `EL_N 15000 >= 1`, you will never see electrons below 25 [GeV](#). On the contrary, If you set `ElectronPt 15000`, you can have for example a higher cut like `EL_N 25000 >= 1` requiring at least one of the electron to be above 25 [GeV](#) in some of the selections, but maybe in other selection you want to use electrons with pt between 15 and 25 GeV (for selection, or for veto). The other important aspect is the `OverlapRemoval`: this is a complicated procedure that is meant to remove double counting of objects: for example, an electron is made from energy clusters in the calorimeter, which can also be reconstructed as a jet, and we don't want that. If an electron and a jet are very near, one of them is removed from the event and not considered anymore. The full procedure in `AnalysisTop` is explained on this page: [OR info](#). The important thing to understand is that all objects satisfying the **object definitions** are used for the OR procedure, even if then in the selections you use much higher cuts: the object definitions are the part where you declare to the code what objects are for you, and therefore what should be

used for the Overlap Removal procedure. We used electrons here as an example, but basically the same arguments apply to all physics objects.

```
BTaggingWP DL1r:FixedCutBEff_77
```

This is where you specify the b-taggers and b-tagging working points you intend to use in your analysis. You need to put a list of strings separated by spaces; the first part of each string is the tagger, and the second part, separated by a `:`, gives the working point. If the working points are available in the b-tagging recommendations, you should get the tagging decision for each jet in branches of the form `jet_isbtagged_something` (or if it's pseudo-continuous calibration, you would get a branch `jet_tagWeightBin_something`). Note that you cannot use a tagger or working point in your event selection if it is not available in the recommendations. For the working points which are calibrated, you will also get event scale-factors (branches of the form `weight_bTagSF_something`). Try to run with, for example:

```
BTaggingWP DL1r:FixedCutBEff_60 DL1r:Continuous
```

uh, wait what? This crashes! Well, the error tells you that we're doing something wrong here.

```
Package.TopEventSelect...ERROR /build/atnight/localbuilds/nightlies/21.2/athena/PhysicsAnalysis/TopPhys/xA0D/TopEventSelectionTools/Root/NJetBtagSele
B-tagging working point DL1r:FixedCutBEff_77 doesn't seem to be supported.
```

Please note that you should provide the argument as `==> bTagAlgorithm:bTagWP` now.

Please provide a real one! Did you specify it in the "BTaggingWP" field of your cutfile?

```
terminate called after throwing an instance of 'std::runtime_error'
  what(): NJetBtagSelector: Invalid btagging selector WP: JET_N_BTAg DL1r:FixedCutBEff_77 >= 1
Aborted (core dumped)
```

Ok, in the event selection we're using the DL1r tagger at the [FixedCutBEff\\_77](#) working point, but we're not configuring it! Let's add it back then:

```
BTaggingWP DL1r:FixedCutBEff_77 DL1r:FixedCutBEff_60 DL1r:Continuous
```

Ok, now it works. We're still using DL1r:FixedCutBEff\_77 only in our selections, but in the output file we have also b-tagging variables relative to the other taggers!

```
root [4] nominal->Print("jet*DL1r*")
*****
*Tree      :nominal      : tree                                     *
*Entries   :      66 : Total =      440757 bytes  File Size =    183442 *
*          :          : Tree compression factor =    1.71             *
*****
*Br        0 :jet_isbtagged_DL1r_77 : vector<char>                  *
*Entries   :      66 : Total Size=      1877 bytes  File Size =      487 *
*Baskets   :       1 : Basket Size=    40960 bytes  Compression=    2.72 *
*.....*
*Br        1 :jet_isbtagged_DL1r_60 : vector<char>                  *
*Entries   :      66 : Total Size=      1877 bytes  File Size =      497 *
*Baskets   :       1 : Basket Size=    40960 bytes  Compression=    2.66 *
*.....*
*Br        2 :jet_tagWeightBin_DL1r_Continuous : vector<int>        *
*Entries   :      66 : Total Size=      2835 bytes  File Size =      652 *
*Baskets   :       1 : Basket Size=    40960 bytes  Compression=    3.43 *
*.....*
...

```

We'll come back to the b-taggers later.

```
TruthCollectionName TruthParticles
TruthJetCollectionName AntiKt4TruthDressedWZJets
TruthLargeRJetCollectionName None # can be AntiKt10TruthTrimmedPtFrac5SmallR20Jets
TopPartonHistory False # can be e.g. ttbar
TopPartonLevel False #this option only exists for recent releases; it switches on/off writing the "truth" parton-level tree
TopParticleLevel False #this switches on/off writing the "particleLevel" tree
TruthBlockInfo False
PDFInfo True
```

These are options which handle the truth contents of your MC sample: when you run on MC, you have the truth record available, and you can access particle-level and parton-level information. `TruthCollectionName` is the name of the truth particles container, and usually you should not need to modify it. `TruthJetCollectionName` and `TruthLargeRJetCollectionName` specify the names of the truth small-R and large-R jet collections, respectively. Some info on truth jets collections are available [here](#). `TopParticleLevel` turns on/off writing the `particleLevel` tree in the output, which contains some particle-level variables for events passing particle-level selections (yes, you can also define particle-level selections in `AnalysisTop`). `TopPartonLevel` is a similar switch for the `truth` tree, which has parton-level information. `TopPartonHistory` is a switch that allows to write in the `truth` tree some useful parton-level information. We'll not go in details on that in this tutorial, but all these options are documented in mode detail [here](#).

## 2.5.2 Systematic modifications



```

### Systematics you want to run
Systematics Nominal #Nominal->only nominal. All -> all systematics (will be much slower). You can also use AllMuons, AllJets... see https://twiki.cern

[...]

#####Reco-level systematics models#####
ElectronEfficiencySystematicModel TOTAL
EgammaSystematicModel 1NP_v1 #egamma calibration uncertainties model
JetUncertainties_NPModel CategoryReduction #JES
JetJERSmearingModel Simple #JER

```

Up to now, we only processes nominal events, but if you want to run over all systematics, you may try `Systematics All`. If you want to only have the systematics associated to a given object, this is possible - try for example `Systematics AllElectrons,AllJets` or `Systematics AllMuons,AllSmallRJets`. It is also possible to specify only one components. There are also options such as `JetUncertainties_NPModel` which handle which systematic scenario we'll use for the different objects (for example with `JetUncertainties_NPModel AllNuisanceParameters` and `Systematics AllSmallRJets` you should see many more trees in your output...). More information on available systematic models can be found [here](#) and [here](#). But to find information on CP systematic uncertainties the best way is of course to read the relevant twiki pages from the CP groups.

Systematics will appear as additional separate trees: for example, `CategoryReduction_JET_Pileup_RhoTopology__1down` and `1up` are two separate trees, which represent what happens when this particular nuisance parameter (NP) of the jet calibration is moved down/up by 1 sigma. You will easily see there are A LOT of systematic uncertainties... Weight systematics (e.g. uncertainties on efficiencies SFs) will instead appear as additional weights in the output ntuple, as already explained above.

Of course running with all systematics on makes the execution MUCH slower, you may want to run on 100 events if you just want to make a quick test.

## 2.5.3 More modifications of the event-level selections

For what regard the selections, they are introduced by the keyword `SELECTION` followed by the name of the selection on the same line, and by a list of cuts. The name of the selection defines the name of the directory in which the cut-flow histograms will be stored, as well as the name of the boolean branch which tells if the event passes the selection or not in the `nominal` tree (e.g. `ejets_2016` above). For what concerns the cuts, the standard event selectors are defined in the [TopEventSelectionTools](#) package; you can find the full list [here](#) in the code (you will learn later how to create you own cut).

But let's have a look, for example, at how the `ejets` selections are defined. In `my-cuts.txt` you should have the following:

The actual selections are defined by the keyword `SELECTION`. As already explained, the fragments introduced by the keyword `SUB` are sub-selections which are used several times in different selections. The subselections are "copied" in the actual selections with a ".". Taking for example the `ejets_DL1r_2016`:

```

SELECTION ejets_DL1r_2016
. EL_2016
. ejets_basic
JET_N_BTAG DL1r:FixedCutBEff_77 >= 1
SAVE

```

We have first the `EL_2016` sub-selection. Let's look at it: it's a simple selection aimed at selecting events which would pass a single electron trigger suitable for 2016 data:

```

SUB EL_2016
. BASIC
. period_2016
GTRIGDEC
EL_N 27000 >= 1

```

`BASIC` contains a serie of "mandatory" cuts for selections which act at reconstructed level: `INITIAL` that sets the start of the cutflow, `GRL` which requests events to be included in the good run list (for data samples), `GOODCALO` which ensures that the LAr and Tile are working properly for the event (no noise bursts), `PRIVTX` which requests at least one primary vertex in the event (this is a very important cut, as events with no PV may cause a crash in CP tools and therefore in your execution), `RECO_LEVEL` means that the selections including `BASIC` are reconstructed-level selection. Unless you really know what you're doing, these options should really be included in all reco-level selections.

```

SUB BASIC
INITIAL
GRL
GOODCALO
PRIVTX
RECO_LEVEL

```

Let's go back looking at our sub-selection `SUB EL_2016`. After including `BASIC`, it includes the `period_2016` subselection. The `period` subselections are defined as:

```

SUB period_2015
RUN_NUMBER >= 276262
RUN_NUMBER <= 284484

SUB period_2016
RUN_NUMBER >= 296939
RUN_NUMBER <= 311481

SUB period_2017
RUN_NUMBER >= 324320
RUN_NUMBER <= 341649

SUB period_2018
RUN_NUMBER >= 348197

```

For example `period_2016` selects events from 2016 data (or MC events which have 2016 data pile-up conditions).

Going back again at the `EL_2016` sub-selection, the next step is `GTRIGDEC`, the global trigger decision. Using the global trigger approach is the recommended way to go for analyses using lepton-based triggers. We configured that and the trigger we want to use in our config file in these lines:

```

### Global lepton trigger scale factor example
UseGlobalLeptonTriggerSF True
GlobalTriggers 2015@e24_lhmedium_L1EM20VH_OR_e60_lhmedium_OR_e120_lhloose,mu20_iloose_L1MU15_OR_mu50 2016@e26_lhtight_nod0_ivarloose_OR_e60_lhmedium_f
GlobalTriggersLoose 2015@e24_lhmedium_L1EM20VH_OR_e60_lhmedium_OR_e120_lhloose,mu20_iloose_L1MU15_OR_mu50 2016@e26_lhtight_nod0_ivarloose_OR_e60_lhme

```

You can see that here we specify which triggers we use for each year. Refer to [this twiki](#) for the most up-to-date recommendations. Finally we have an appropriate offline electron pT cut, requiring to have at least one electron with  $p_T > 27$  GeV in events to be saved.

Ok, so we now understood that `EL_2016` is a very basic sub-selection which requires clean events, passing the trigger we want to use and with at least one electron with  $p_T > 27$  GeV for 2016 data events or for events replicating 2016 conditions in MC, and the same is true for the similar selections `EL_2015`, `EL_2017`, ... for the other data-taking years.

Let's look now at the `SELECTION ejets_DL1r_2016` final selection.

```

SELECTION ejets_DL1r_2016
. EL_2016
. ejets_basic
JET_N_BTAG DL1r:FixedCutBEff_77 >= 1
SAVE

```

This is including our very simple selection `EL_2016`, so we will have only events passing that one.

Then it includes the `ejets_basic` sub-selection: have a look at that in your config file, you will see that this includes a list of kinematic cuts on leptons (exactly one electron with  $p_T > 25$  [GeV](#), exactly no muon with  $p_T > 25$  [GeV](#)), then it requires `GTRIGMATCH`: the electron in the events must have been the one firing one of the triggers you configured. Then with `JETCLEAN LooseBad` we set the jet cleaning level we want, then we select how many jets we want in our events (at least 4 with  $p_T > 25$  [GeV](#) in the end, but we do that in steps to keep the information in the cutflow). Then we apply cuts on the MET, MTW. With `RECO::KLFITERRUN kElectron` `KLFFitterBTaggingWP:DL1r:FixedCutBEff_77` we say that we want to use the KLFFitter (if the line is uncommented), with which tagger and at which working point. `#KLFFITTER > -50.0` is commented, but we may want to cut on the KLFFitter score by using something like that. `EXAMPLEPLOTS` will just produce some plots in the cutflow directory for our selection. `NOBADMUON` will remove few events which may have a pathological muon.

Then we're including a cut on the number of b-jets tagged from DL1r at the 77% efficiency working point.

Finally, the `SAVE` keyword which asks for saving events passing this selection if it has reached this stage. You may have other cuts after the `SAVE` keyword, which won't prevent the event to be saved if they fail these additional cuts (but we would see their effect in the cut-flows). Also, if you don't specify the `SAVE` keyword, the event will be saved only if it passes at least one other selection.

Do you want to play a bit? Try for example to add another selection:

```

SELECTION ejets_DL1rveto_2016
. EL_2016
. ejets_basic
JET_N_BTAG DL1r:FixedCutBEff_77 < 1
SAVE

```

What is this doing? You can define also a selection with no requirements on the b-tagging, or just basically anything you want. As usual, refer to the [Start guide](#) for more information on what you can do with the selections.

## 2.5.4 Using particle-level selections

Again on the `my-cuts.txt` file, let's now have a look at particle-level stuff. First of all, let's give again a different name to the output file, let's run on Nominal systematics only and let's comment out the KLFFitter lines. Also, let's run on 500 events and, most important of all, turn-on the particle-level stuff:

```

NEvents 500
[...]
OutputFilename output-modifiedSelection_PL.root
[...]
Systematics Nominal
[...]
TopParticleLevel True
[...]
#RECO::KLFITERRUN kElectron KLfitterBTaggingWP:DL1r:FixedCutBEff_77
[...]
#RECO::KLFITERRUN kMuon KLfitterBTaggingWP:DL1r:FixedCutBEff_77

```

Run as usual with `top-xaod my-cuts.txt input-mc-rel21_p4031.txt`, then have a look at the output file:

```

root -l output-modifiedSelection_PL.root
[] .ls
[] particleLevel->GetEntries()

```

You can see that this tree is now filled, isn't it? Play a bit with it, plot some of its content to familiarise with it. You can see we have some basic event-level stuff, kinematic quantities for particle-level stuff, PDF information (because we set `PDFInfo True` in the config file, and as reported [here](#) this means we're writing PDF info in the truth and particle-level trees), and that we then have the booleans for the selections. Now, the ones for reco-level selections (like `mujets_DL1r_2016`) are always 0 (of course, we're looking at particle-level here), but `ejets_particle` and `mujets_particle` are filled. You can check at the end of the config file how we define those particle-level selections, and if you want play a bit with that. Documentation on available selectors is as usual in [the start guide](#).

## 2.6 Browsing the AnalysisTop code

We already mentioned this above, but it's worth discussing it again. At some point you will want to have a look at the AnalysisTop code, to get some inspiration on how to write some code, to understand how a variable is filled, or to maybe check if you need some changes in the central AT code.

You can find [here](#) all the AT packages: it's a good idea to bookmark this link. You should see something like this:

atlas > athena > Repository

21.2

athena / PhysicsAnalysis / TopPhys / xAOD

History

Find file

Web IDE

Download

Clone

/ +

Merge branch '21.2-MyAnalysisTopUpdate\_newOptions' into '21.2'

...

67f6de1c

Nils Erik Krumnack authored 6 days ago

Name	Last commit	Last update
..		
TopAnalysis	Merge branch '21.2-MyAnalysisTopUpdate_newOptions' into '21...	6 days ago
TopCPTools	Merge branch '21.2-MyAnalysisTopUpdate_newOptions' into '21...	6 days ago
TopConfiguration	Merge branch '21.2-MyAnalysisTopUpdate_newOptions' into '21...	6 days ago
TopCorrections	update of copyright statements	4 weeks ago
TopEvent	AnalysisTop: Deprecate cout use in favor of ATH_MSG	1 month ago
TopEventReconstructionTools	AnalysisTop: fixing the name of the KLfitter tool	3 weeks ago
TopEventSelectionTools	Replacing reco - truth / truth distributions with reco/truth	1 week ago
TopExamples	Adding new shower algorithms to the list, needed for MC/MC SFs	4 weeks ago
TopHLUpgrade	Fixing TopHLUpgrade	5 months ago
TopJetSubstructure	fix warnings/errors with clang on MacOS	1 month ago
TopObjectSelectionTools	AT Add support for time-stamped track-jet collections for b-tag...	1 month ago
TopParticleLevel	AnalysisTop: Deprecate cout use in favor of ATH_MSG	1 month ago
TopPartons	fix warnings/errors with clang on MacOS	1 month ago
TopSystematicObjectMaker	Merge branch '21.2-MyAnalysisTopUpdate_1903' into '21.2' Anal...	2 weeks ago

**One very important thing is the dropdown menu showing 21.2 in this image.** This is to select the code for a given release. When "21.2" (which currently is the default branch we use, DON'T LOOK AT THE "master" branch) is selected, you're looking at the HEAD, i.e. the very last version available for our code. If you want to check for the code of a particular release (e.g. because you're using that release in your analysis and you want to check what is being done exactly in that release), click on that menu and search for the release you want, e.g. "21.2.119".

A couple of files you may need to visit relatively often in gitlab are:

- `TopConfiguration/Root/ConfigurationSettings.cxx` -> as we mentioned above, you find here all config options you can use and their default values
- `TopAnalysis/Root/EventSaverFlatNtuple.cxx` -> this is the standard saver used to write the AT output

The packages of AnalysisTop are described in more detail [in this twiki](#).

## 3 Extending AnalysisTop

After playing with the configuration file options, you may want to make more complicated modifications, for which you may have to create your own extension of AnalysisTop. To do this, you can create an analysis package, or a group of packages, which are compiled within an AnalysisTop release. This is basically what projects like TTHbbAnalysis or ttHMultiAna are. In this section, you will retrieve from git such a small project to learn how to make such kind of extension, using a small package called [HowToExtendAnalysisTop](#) and located in the [atlasphys-top/reco](#) repository.

This package is intended to be for demonstrative purpose only.

### 3.1 Setting up the HowToExtendAnalysisTop package

We'll now use [this repository](#) which includes a full example on how to extend AnalysisTop.

Let's start from a fresh shell, do the standard atlas setup and setup git, then go to your work directory and clone the repository:

```
setupATLAS
lsetup git
cd ~/work/MyAnalysis #or whatever is the path where you prepared your area for the first part of the tutorial
cd source
git clone ssh://git@gitlab.cern.ch:7999/atlasphys-top/reco/howtoextendanalysistop.git
```

You should now see the directory `howtoextendanalysistop` inside the athena folder. Note that the setup of git by `lsetup git` will only work if you have done `setupATLAS` before. Also, git won't work properly if you don't use `lsetup git` (the git setup we get from lxplus directly isn't the same as the one we get from ATLAS).

Then, you can setup AnalysisBase, and perform the compilation as usual from your work directory, using the scripts we prepared above:

```
cd ..
source setup.sh
source full_compile.sh #we need a full compilation because we added a new package
```

### 3.2 Creating a custom-saver

This is by far the most common thing you'll have to do. AnalysisTop does a lot of stuff for you, but in the end you will want to add some new variables which are not stored by default. This is very simple using a custom saver! For the tutorial, we'll use the `howtoextendanalysistop` package. For the future, if you need at some point to create your own custom saver from scratch, you may be interested also in [this automatic approach](#): this can save you some time.

#### 3.2.1 The very basic

Let's start with some clean options again, after we played so much with the previous ones. Go in the `run` directory and do:

```
cp $AnalysisBase_DIR/data/TopAnalysis/example-config.txt custom-saver-test.txt
```

then open `custom-saver-test.txt` in your favorite editor. Open also the relevant files from the `source` directory:

`../source/howtoextendanalysistop/HowtoExtendAnalysisTop/CustomEventSaver.h` and `../source/howtoextendanalysistop/Root/CustomEventSaver.cxx`. You will that the header has a class declaration:

```
class CustomEventSaver : public top::EventSaverFlatNtuple
```

`CustomEventSaver` is our custom saver, and it inherits from the standard one for [AnalysisTop](#), `EventSaverFlatNtuple` (you can browse that one at some point to get an idea, as explained above, but it's a looong one). The idea is that this saver will do everything that the `EventSaverFlatNtuple` can do, but add some cherries on top. In the header you can also see that we declare two new (private) variables:

```
float m_randomNumber;
float m_someOtherVariable;
```

In this simple example these are two variables we want to store in our output ntuple.

Also notice another couple of things in the header:

```

#ifndef HOWTOEXTENDANALYSISTOP_CUSTOMOBJECTLOADER_H
#define HOWTOEXTENDANALYSISTOP_CUSTOMOBJECTLOADER_H

[...]

namespace top {

[...]

}

#endif

```

the "ifndef -> define -> endif" block is something which you're hopefully familiar with, if not google a bit because it's important: basically it's a protection that avoids that makes the code compile only once, even if the class becomes at some point included by several other files: if you don't do this, you could get compilation errors. It's important that the name you use after the ifndef and define is one representative of your class and which is absolutely unlikely to be used somewhere else in the whole code, otherwise you will have problems. So, using a format like NAMEOFTHEPACKAGE\_NAMEOFTHEFILE makes sense. When you will add new files to the code, remember to use this technique. The other thing to notice is that we're working within the `top` namespace: this is because we're working within AnalysisTop, and we use that namespace in AT. Not familiar with namespaces in C++? Again, please google.

Let's now look at the .cxx file.

```

CustomEventSaver::CustomEventSaver() :
    m_randomNumber(0.),
    m_someOtherVariable(0.)
{
}

```

In the constructor we initialize those variables to a sensible number; this happens only the very first time the constructor is created, before AT reads the first event from the input, and is not executed anymore then. Then

```

//-- initialize - done once at the start of a job before the loop over events --//
void CustomEventSaver::initialize(std::shared_ptr<top::TopConfig> config, TFile* file, const std::vector<std::string>& extraBranches)
{
    //-- Let the base class do all the hard work --//
    //-- It will setup TTrees for each systematic with a standard set of variables --//
    top::EventSaverFlatNtuple::initialize(config, file, extraBranches);

    //-- Loop over the systematic TTrees and add the custom variables --//
    for (auto systematicTree : treeManagers()) {
        systematicTree->makeOutputVariable(m_randomNumber, "randomNumber");
        systematicTree->makeOutputVariable(m_someOtherVariable, "someOtherVariable");
    }
}

```

as for the comment, this also is executed only once at the beginning. With these instructions we're first setting up the standard EventSaverFlatNtuple stuff, and then we're adding to ALL trees (the nominal and the systematic ones) new branches where we'll store the new variables we're adding to the trees. Finally:

```

//-- saveEvent - run for every systematic and every event --//
void CustomEventSaver::saveEvent(const top::Event& event)
{
    //-- set our variables to zero --//
    m_randomNumber = 0.;
    m_someOtherVariable = 0.;

    //-- Fill them - you should probably do something more complicated --//
    TRandom3 random( event.m_info->eventNumber() );
    m_randomNumber = random.Rndm();
    m_someOtherVariable = 42;

    //-- Let the base class do all the hard work --//
    top::EventSaverFlatNtuple::saveEvent(event);
}

```

This method runs on all events which will be written in the output (usually, this means all events **passing at least one of the selections**). The const reference to a [TopEvent](#) object `event` contains all the information we have access to. You can [browse how this class looks like](#) if you want to; we'll see something on that topic later. As you can see in this method we first set an initial value to the 2 new variables we want to write in the output ntuple, then we fill them (one with a random number and one with the answer to life, the universe and everything, i.e. 42). After that we're done, we simply pass the event to the method of the base class and leave all the dirty work to it.

If you didn't do a full compilation, do it now, and then we have to adjust the config file to use our custom saver instead of the standard one: first of all, we have to add the library corresponding to our new package to the ones that are loaded

```

LibraryNames libTopEventSelectionTools libTopEventReconstructionTools libHowToExtendAnalysisTop

```

and then do:

```
#OutputFormat top::EventSaverFlatNtuple
OutputFormat top::CustomEventSaver
```

in your `custom-saver-test.txt` config file, and run `top-xaod custom-saver-test.txt input-mc-rel21_p4031.txt`. Look at the output: do you see our two new variables there?

### 3.2.2 Let's add something real to the output now

Let's do some real work now. The derivations we use in input have more information than the ones stored by default by AnalysisTop. Let's have a look! Do a `cat input-mc-rel21_p4031.txt` to check what is the file we've been using until now, and open that with root. Now, this is a more complicated format, and you may see some complaints from root, but ignore them. You can see that there are several helper trees (e.g. the ones containing metadata for the sample) but ignore them, and have a look at `CollectionTree`: this is the one containing our precious data.

If you have a look at that, it has A LOT of branches, relative to the event or to the objects associated with the events. Let's focus for the time being on the jets: you can see there are many different kind of jets. We're using `antiKt4EMPFLOW` jets (this is how we're configuring jets in the config file, no?), so let's have a look at them. In the root terminal do:

```
[ ] CollectionTree->Print("AntiKt4EMPFLOW*")
```

still a lot of stuff. Let's pretend now that for our analysis for some reason we want to store the EM fraction (i.e. the fraction of the energy of the jet released in the electromagnetic calorimeter). You may notice the associated branch in the long output of the last command; if not, just do

```
[ ] CollectionTree->Print("AntiKt4EMPFLOW*EMFrac*")
*****
*Tree   :CollectionTree: CollectionTree                               *
*Entries :    60000 : Total =    37349097338 bytes  File Size = 8225265562 *
*       :           : Tree compression factor =    4.54                  *
*****
*Br      0 :AntiKt4EMPFLOWJetsAuxDyn.EMFrac : vector<float>              *
*Entries :    60000 : Total Size=    2930713 bytes  File Size =    2228544 *
*Baskets :     653 : Basket Size=     5632 bytes  Compression=    1.31    *
*.....*
```

Ok, so this is a vector of floats: well, this is expected, for each jet we have an associated EM fraction, so we have a vector, with one entry per jet. How can we add this to the output of AT? Of course with our [CustomSaver](#).

First of all, open the `HowtoExtendAnalysisTop/CustomEventSaver.h` header. We have to add a variable to hold our new variable:

```
private:
    ///-- Some additional custom variables for the output ---//
    float m_randomNumber;
    float m_someOtherVariable;
    std::vector<float> m_jet_EMFrac;
```

**Important NOTE: cure your indentation.** At some point you may want to include some of your code in the central AT, or to share it with other people, e.g. So, make sure the indentation you are using is correct: in AnalysisTop each indentation level corresponds to **2 spaces**, and it's a **very good** idea to configure your editor to do this automatically. Also, try to use reasonable names and comments (always in english): even if you are the only one using a code, at some point you may want to share it, or to ask people for help, and they will be able to do that only if they can read it.

Now in the `Root/CustomEventSaver.cxx` file, we have to tell the Saver we want to add this to the output tree:

```
///-- initialize - done once at the start of a job before the loop over events ---//
void CustomEventSaver::initialize(std::shared_ptr<top::TopConfig> config, TFile* file, const std::vector<std::string>& extraBranches)
{
    ///-- Let the base class do all the hard work ---//
    ///-- It will setup TTrees for each systematic with a standard set of variables ---//
    top::EventSaverFlatNtuple::initialize(config, file, extraBranches);

    ///-- Loop over the systematic TTrees and add the custom variables ---//
    for (auto systematicTree : treeManagers()) {
        systematicTree->makeOutputVariable(m_randomNumber, "randomNumber");
        systematicTree->makeOutputVariable(m_someOtherVariable, "someOtherVariable");
        systematicTree->makeOutputVariable(m_jet_EMFrac, "jet_EMFrac");
    }
}
```

What if you want to save some variables only in the nominal tree, but not in all the systematics variations, to save space? You could e.g. do `if(systematicTree->name()=="nominal") systematicTree->makeOutputVariable(m_jet_EMFrac, "jet_EMFrac");` instead in this case; but let's not do that, just keep it in mind. Then let's do:



```

///-- saveEvent - run for every systematic and every event --///
void CustomEventSaver::saveEvent(const top::Event& event)
{
    ///-- set our variables to zero --///
    m_randomNumber = 0.;
    m_someOtherVariable = 0.;
    m_jet_EMFrac.clear();

    [...]

```

First of all, we have to clear the `m_jet_EMFrac` vector at the beginning of each event. There are other ways of doing it (e.g. by resizing the vector in each event...) but we'll stick with this way for simplicity. The important thing is to make sure that normal variables are initialized properly for each event (you don't want to end in weird cases where you keep the value from the previous event) and that you cure vector variables in a way you're not just adding new stuff to them in each event until they explode. If we start adding a lot of variables, we may want to split their per-event initialization in a dedicated method, so let's declare that in the header:

```

...
private:
    ///-- Some additional custom variables for the output --///
    float m_randomNumber;
    float m_someOtherVariable;
    std::vector<float> m_jet_EMFrac;

    void initEvent();
...

```

and then move some code in the `.cxx` file.

```

///-- initEvent - initializes output variables at the beginning of each event --///
void CustomEventSaver::initEvent()
{
    m_randomNumber = 0.;
    m_someOtherVariable = 0.;
    m_jet_EMFrac.clear();
}

///-- saveEvent - run for every systematic and every event --///
void CustomEventSaver::saveEvent(const top::Event& event)
{
    ///-- set our variables to zero --///
    initEvent();

    ///-- Fill them - you should probably do something more complicated --///
    TRandom3 random( event.m_info->eventNumber() );
    m_randomNumber = random.Rndm();
    ...

```

Now the initialization part is done, let's really fill this variable:

```

...
///-- saveEvent - run for every systematic and every event --///
void CustomEventSaver::saveEvent(const top::Event& event)
{
    ///-- set our variables to zero --///
    initEvent();

    ///-- Fill them - you should probably do something more complicated --///
    TRandom3 random( event.m_info->eventNumber() );
    m_randomNumber = random.Rndm();
    m_someOtherVariable = 42;

    for(const xAOD::Jet* jet : event.m_jets)
    {
        float emfrac=-999.;
        if(jet->isAvailable<float>("EMFrac")) emfrac=jet->auxdata<float>("EMFrac");
        m_jet_EMFrac.push_back(emfrac);
    }

    ///-- Let the base class do all the hard work --///
    top::EventSaverFlatNtuple::saveEvent(event);
}

```

The `for(const xAOD::Jet* jet : event.m_jets)` loop is looping on the jets included in my `top::Event` objects: these are ALL the jets passing the **object-level definitions** you define in the config file. It's a very good idea to have a look [at the code of the top::Event class](#), so you realize what other stuff you can use from that. Ok, now we're looping on jets in the evts, then we just define a variable to hold our EM fraction `float emfrac=-999.;`. Then the trick: `if(jet->isAvailable("EMFrac")) emfrac=jet->auxdata("EMFrac");`. What is happening here is that the EMFrac is included in the Jet object as a so-called **decoration**, with the `isAvailable` method we verify if this is really available (this may not be true for some derivations formats for example, and if we don't check our code will crash there), and then if the answer is yes we retrieve the value with the **auxdata** method. As you can see for both these two methods we have to specify what is the type of the variable, so if you're working with an int instead that with a float you can guess what you have to do... All auxiliary variables in the derivations (remember the

"Aux" in the "AntiKt4EMPFlowJetsAuxDyn.EMFrac" branch?) are accessible like that, and let us repeat: do not just assume all of them are available in all derivation formats, because that's not true 🤖

With the last line `m_jet_EMFrac.push_back(emfrac);` we just fill the vector with the value we retrieved, and we're done.

Run again with `top-xaod custom-saver-test.txt input-mc-rel21_p4031.txt` : can you see the new variable in the output? Try eg. a `nominal->Scan("jet_pt:jet_EMFrac")` in the root terminal.

### 3.2.3 Getting access to the config options and retrieving containers

Let's do something even more complicated now: let's say we want to find the nearest particle-level jet for each reconstructed jet and store the associated truth momentum. There is actually already a nice way of producing some plots for that automatically in AnalysisTop, but here we're using this example to learn something new.

First of all we add the new variable in the header:

```
...
private:
    ///-- Some additional custom variables for the output --///
    float m_randomNumber;
    float m_someOtherVariable;
    std::vector<float> m_jet_EMFrac;
    std::vector<float> m_jet_ptOfAssociatedTruthJet;
...
```

and in the .cxx

```
...
///-- initialize - done once at the start of a job before the loop over events --///
void CustomEventSaver::initialize(std::shared_ptr<top::TopConfig> config, TFile* file, const std::vector<std::string>& extraBranches)
{
    ///-- Let the base class do all the hard work --///
    ///-- It will setup TTrees for each systematic with a standard set of variables --///
    top::EventSaverFlatNtuple::initialize(config, file, extraBranches);

    ///-- Loop over the systematic TTrees and add the custom variables --///
    for (auto systematicTree : treeManagers()) {
        systematicTree->makeOutputVariable(m_randomNumber, "randomNumber");
        systematicTree->makeOutputVariable(m_someOtherVariable, "someOtherVariable");
        systematicTree->makeOutputVariable(m_jet_EMFrac, "jet_EMFrac");
        systematicTree->makeOutputVariable(m_jet_ptOfAssociatedTruthJet, "jet_ptOfAssociatedTruthJet");
    }
}

///-- initEvent - initializes output variables at the beginning of each event --///
void CustomEventSaver::initEvent()
{
    m_randomNumber = 0.;
    m_someOtherVariable = 0.;
    m_jet_EMFrac.clear();
    m_jet_ptOfAssociatedTruthJet.clear();
}
...
```

Now we have to get access to the truth jets. There are some ways of doing that, but let's do a manual retrieve of the truth-jets container. The first thing to do is understanding: how is the container called? We have that name in the config file `TruthJetCollectionName` [AntiKt4TruthDressedWZJets](#). Ok, so we copy and paste it in the code? Nah, that's a bad idea. At some point you may want to run your code on a different derivation format using a different name for the truth-jet collection, for example. If you hard-code it, you'll have to change it in the code and recompile. You can instead just get access to that information from the config file, so you can just change that without recompiling, and be able to run on any collections of truth-jets you want with the same code, just using different config files (for example).

So let's do the following: in the header of our custom saver we have to add:

```
#include "TopConfiguration/TopConfig.h"
...
```

to the includes: the [TopConfig class](#) is the class which manages all the AT options, and contains all the values we configured in our config-file (and the default values for the one we didn't explicitly configure). Remember that <https://gitlab.cern.ch/atlas/athena/-/blob/21.2/PhysicsAnalysis/TopPhys/xAOD/TopConfiguration/Root/ConfigurationSettings.cxx> is instead the place where the names of the options and their default values are stored.

Now, let's add a declaration in the header:

```
private:

    std::shared_ptr<top::TopConfig> m_topconfig;
    ///-- Some additional custom variables for the output ---
    float m_randomNumber;
    float m_someOtherVariable;
    ...
```

you're not scared by [smart pointers](#), are you? (If yes, it's a good idea to google a bit and have a look at them, they can really save your life).

And finally in the .cxx

```
///-- initialize - done once at the start of a job before the loop over events ---
void CustomEventSaver::initialize(std::shared_ptr<top::TopConfig> config, TFile* file, const std::vector<std::string>& extraBranches)
{
    m_topconfig=config;
    ///-- Let the base class do all the hard work ---
    ///-- It will setup TTrees for each systematic with a standard set of variables ---
```

What are we doing here? We're just storing the pointer to the config file we have access to from the `initialize` method in a private member of our class, so now we have access to that everywhere in the class. And we're using a smart pointer, so we don't have to worry about who owns the pointer, who has to delete it, and so on.

Want to test if this works?

In the .cxx, add `a = #include =` (which is a good idea anyway), and then a simple cout:

```
void CustomEventSaver::saveEvent(const top::Event& event)
{
    ///-- set our variables to zero ---
    initEvent();

    std::cout<<"truth jet collection="<<m_topconfig->sgKeyTruthJets()<<std::endl;
```

`sgKeyTruthJets` is the method of the [TopConfig class](#) which stores the name of the collection of the truth-jets. Compile and see if this is printed out! Yes? Good, now remove the cout line and let's proceed.

We need to actually retrieve the truth-jets container. **Be very careful** about manual retrieving of containers: for reco-level objects, this would mean taking the objects directly from the derivation, **without any calibration and correction from the CP tools**. These corrections are applied to the objects used by the AnalysisTop code, for example the ones store in the `top::Event` object (like the jets we accessed in the `for(const xAOD::Jet* jet : event.m_jets)` loop); if you manually retrieve the container, **this is not true**. So do that only if you **really** know what you're doing. For truth objects, there is no calibration to apply, so things are somehow easier.

In the header of our custom saver we now have to include the appropriate container class:

```
#include "xAODJet/JetContainer.h"
```

(how did we know the correct name? look at the code of the [top::Event class](#)). Then, always in the header, declare an appropriate class member:

```
...
private:

    std::shared_ptr<top::TopConfig> m_topconfig;
    const xAOD::JetContainer* m_truthJets;
    ///-- Some additional custom variables for the output ---
    float m_randomNumber;
    float m_someOtherVariable;
    ...
```

try to compile it like that for the time being. Does it work? Ok, now let's play our trick:

```
///-- initEvent - initializes output variables at the beginning of each event ---
void CustomEventSaver::initEvent()
{
    //initialization of variables
    m_randomNumber = 0.;
    m_someOtherVariable = 0.;
    m_jet_EMFrac.clear();
    m_jet_ptOfAssociatedTruthJet.clear();
    //retrieving containers now
    m_truthJets = 0;
    top::check(evtStore()->retrieve( m_truthJets, m_topconfig->sgKeyTruthJets()), "Failed to retrieve truth jets");
}
```

We do the retrieving in the `initEvent` method, so this is done at the beginning of each event. **It's important to empty the pointer before retrieving the container, otherwise you may have a crash**. The `evtStore` is something keeping a lot of important stuff (containers, tools used by other part of the code...) but let's not go in detail now. With the `retrieve` method we take the "`m_topconfig->sgKeyTruthJets()`" container (corresponding to the name for the truth jets we define in the config file) and we store them in the "`m_truthJets`" container. This is done inside a "`top::check`" method, which controls if everything goes fine, otherwise it prints out an error and cause a crash, so we know where to look for problems. Try to compile. Does it work? No? Well, we forgot that in order to use "`top::check`" we need to add

```
#include "TopEvent/EventTools.h"
```

in our code. Does it work now? Ok, we're ready to finish our work:

```
///-- saveEvent - run for every systematic and every event ---//
void CustomEventSaver::saveEvent(const top::Event& event)
{
    ///-- set our variables to zero ---//
    initEvent();

    ///-- Fill them - you should probably do something more complicated ---//
    TRandom3 random( event.m_info->eventNumber() );
    m_randomNumber = random.Rndm();
    m_someOtherVariable = 42;

    for(const xAOD::Jet* jet : event.m_jets)
    {
        float emfrac=-999.;
        if(jet->isAvailable<float>("EMFrac")) emfrac=jet->auxdata<float>("EMFrac");
        m_jet_EMFrac.push_back(emfrac);

        const xAOD::Jet* associated_truth_jet=0; //for each jet we define a pointer to the closest truth-jet, so afterwards we can store evrything we want
        float minDR=0.4; //we only find a corresponding truth-jet if DR(truth-jet,reco-jet)<0.4, otherwise we assume there is no correspondance.
        //THIS IS JUST A VALUE USED FOR THIS EXAMPLE, you should define a reasonable association for a real analysis, and we don't provide a default value

        for(const xAOD::Jet* tjet : *m_truthJets) //we loop on the truth-jets we retrieved in the initEvent method at the beginning of the event
        {
            if(tjet->pt()<10000.) continue; //we cannot look at truth-jets with a too low-pt, it doesn't make sense

            float DR=tjet->p4().DeltaR(jet->p4()); //we calculate the angular distance DR(reco-jet,truth-jet)
            if(DR<minDR) //if the truth-jet is the closest one, we store it in the pointer we prepared (and we store the minimum DR found until now)
            {
                minDR=DR;
                associated_truth_jet=tjet;
            }
        }
        //end of loop on truth jets
        m_jet_ptOfAssociatedTruthJet.push_back(associated_truth_jet ? associated_truth_jet->pt() : -999); //finally we fill the output vector; remember
        //the pointer could be 0 if we found no truth-jet within DR=0.4 from the reco-jet
        //the ? : construct is equivalent to a (longer)
        //float tjet_pt=-999;
        //if(associated_truth_jet) tjet_pt=associated_truth_jet->pt();
        //else tjet_pt=-999;
        //m_jet_ptOfAssociatedTruthJet.push_back(tjet_pt);

    }
    //end of loop on jets
    ///-- Let the base class do all the hard work ---//
    top::EventSaverFlatNtuple::saveEvent(event);
}
```

Hopefully the comments in this code are enough to explain what we are doing. You're ready to test the code: compile and run it as usual `top-xaod custom-saver-test.txt input-mc-rel21_p4031.txt`. Open the output file and check it in root:

```
[ ] nominal->Scan("jet_pt:jet_ptOfAssociatedTruthJet")
```

Ok, you are ready to perform jet resolution and scale studies if you want to!

### 3.2.4 Using dynamic keys

One somehow disturbing thing in the code above is that we hard-coded the maximum DR allowed for the truth-reco jet association and the pt cut on truth jets. Ah, if only we could set them in the config file as we do for other things... Well, we actually can. Using the so-called dynamic keys.

In the header, let's add variables to store the values we'll read from the config file:

```
...
private:

    float m_RecoTruthJetAssociation_DRCut;
    float m_RecoTruthJetAssociation_minTruthJetPtCut;
...

```

In the cxx file, add :

```
#include "TopConfiguration/ConfigurationSettings.h"
```

and take care of initializing the cuts to reasonable values

```

//--- Construtor ---//
CustomEventSaver::CustomEventSaver() :
    m_RecoTruthJetAssociation_DRCut(0.4),
    m_RecoTruthJetAssociation_minTruthJetPtCut(10000.),
    m_randomNumber(0.),
    m_someOtherVariable(0.)
{
}

```

Then we read the two options from the config-file in the `initialize` method:

```

void CustomEventSaver::initialize(std::shared_ptr<top::TopConfig> config, TFile* file, const std::vector<std::string>& extraBranches)
{
    top::ConfigurationSettings* configSettings = top::ConfigurationSettings::get(); //this can be used in case we need dynamic keys

    try{
        m_RecoTruthJetAssociation_DRCut = std::stof(configSettings->value("RecoTruthJetAssociationDRCut"));
    }
    catch (...) {
        throw std::invalid_argument {
            "TopConfig: cannot convert Option RecoTruthJetAssociationDRCut into float"
        };
    }
    try{
        m_RecoTruthJetAssociation_minTruthJetPtCut = std::stof(configSettings->value("RecoTruthJetAssociationMinTruthJetPtCut"));
    }
    catch (...) {
        throw std::invalid_argument {
            "TopConfig: cannot convert Option RecoTruthJetAssociationMinTruthJetPtCut into float"
        };
    }
    ...
}

```

We have the small inconvenience that the "value" method returns a string, and we need a float, so we use the C++ function `std::stof` to do the string->float conversion, and we use a try...catch statement to protect the code and give a clear error in case by mistake we give a non-float value to the option (if you're not familiar with the try-catch construct, google a bit). Finally we use them in the `saveEvent` method:

```

const xAOD::Jet* associated_truth_jet=0; //for each jet we define a pointer to the closest truth-jet, so afterwards we can store evrything we want
float minDR=m_RecoTruthJetAssociation_DRCut; //we only find a corresponding truth-jet if DR(truth-jet,reco-jet)<0.4, otherwise we assume there
//THIS IS JUST A VALUE USED FOR THIS EXAMPLE, you should define a reasonable association for a real analysis, and we don't pr

for(const xAOD::Jet* tjet : *m_truthJets) //we loop on the truth-jets we retrieved in the initEvent method at the beginning of the event
{
    if(tjet->pt(<m_RecoTruthJetAssociation_minTruthJetPtCut) continue; //we cannot look at truth-jets with a too low-pt, it doesn't make sense

    float DR=tjet->p4().DeltaR(jet->p4()); //we calculate the angular distance DR(reco-jet,truth-jet)
    if(DR<minDR) //if the truth-jet is the closest one, we store it in the pointer we prepared (and we store the minimum DR found until now)
    {
        minDR=DR;
        associated_truth_jet=tjet;
    }
}
//end of loop on truth jets

```

Now, modify your config file by declaring the two new options "RecoTruthJetAssociation\_DRCut" and "RecoTruthJetAssociation\_minTruthJetPtCut" as dynamickeys and assign them the value you want

```

DynamicKeys RecoTruthJetAssociationDRCut,RecoTruthJetAssociationMinTruthJetPtCut
RecoTruthJetAssociationDRCut 0.3
RecoTruthJetAssociationMinTruthJetPtCut 15000.

```

Of course, it would make sense to save some more variables (e.g. the DR distance between the jets...), feel free to try by yourself.

### 3.2.5 Adding some particle-level stuff

What if we want to add something to the `particleLevel` tree? For example, we notice that in that tree we don't store by default labels corresponding to the origin of the truth jet (you'll have to run with `TopParticleLevel True` of course to see the tree). It's quite simple: first we add a `saveParticleLevelEvent` declaration in the header of the custom saver and we include the "TopParticleLevel/ParticleLevelEvent.h" header:

```

#include "TopParticleLevel/ParticleLevelEvent.h"
...
//--- We will be setting our custom variables on a per-event basis ---//
virtual void saveEvent(const top::Event& event) override;
virtual void saveParticleLevelEvent(const top::ParticleLevelEvent& plEvent) override;

private:
...

```

and in the .cxx

```
void customEventsaver::saveParticleLevelEvent(const top::ParticleLevelEvent& pEvent) {

    top::EventSaverFlatNtuple::saveParticleLevelEvent(pEvent);
} //end of saveParticleLevelEvent
```

This is the particle-level equivalent of saveEvent, calling the default saveParticleLevelEvent at the end to have that doing the hard work. Now, back in the header we declare the variables we want to store:

```
...
    ///-- Some additional custom variables for the output ---//
    float m_randomNumber;
    float m_someOtherVariable;
    std::vector<float> m_jet_EMFrac;
    std::vector<float> m_jet_ptOfAssociatedTruthJet;

    ///--output variables for the pl tree ---//
    std::vector<int> m_pltree_jet_HadronConeExclExtendedTruthLabelID;
    std::vector<int> m_pltree_jet_HadronConeExclTruthLabelID;
    std::vector<int> m_pltree_jet_PartonTruthLabelID;
...

```

we tell the tree manager we want to save them inside the initialize method:

```
...
    ///-- Loop over the systematic TTrees and add the custom variables ---//
    for (auto systematicTree : treeManagers()) {
        systematicTree->makeOutputVariable(m_randomNumber, "randomNumber");
        systematicTree->makeOutputVariable(m_someOtherVariable, "someOtherVariable");
        systematicTree->makeOutputVariable(m_jet_EMFrac, "jet_EMFrac");
        systematicTree->makeOutputVariable(m_jet_ptOfAssociatedTruthJet, "jet_ptOfAssociatedTruthJet");
    }

    if(config->doTopParticleLevel())
    {
        if ( not particleLevelTreeManager() )
            EventSaverFlatNtuple::setupParticleLevelTreeManager();
        particleLevelTreeManager()->makeOutputVariable(m_pltree_jet_HadronConeExclExtendedTruthLabelID, "jet_HadronConeExclExtendedTruthLabelID");
        particleLevelTreeManager()->makeOutputVariable(m_pltree_jet_HadronConeExclTruthLabelID, "jet_HadronConeExclTruthLabelID");
        particleLevelTreeManager()->makeOutputVariable(m_pltree_jet_PartonTruthLabelID, "jet_PartonTruthLabelID");
    }
...

```

and we fill them in the saveParticleLevelEvent, accessing the decorations as we learnt already, taking care of clearing the vectors at the beginning of each event (if you want to do that in a separate dedicated method, called e.g. initPLEvent, feel free to do that):

```
void customEventsaver::saveParticleLevelEvent(const top::ParticleLevelEvent& pEvent) {

    m_pltree_jet_HadronConeExclExtendedTruthLabelID.clear();
    m_pltree_jet_HadronConeExclTruthLabelID.clear();
    m_pltree_jet_PartonTruthLabelID.clear();

    for(const xAOD::Jet* jet : *pEvent.m_jets)
    {
        int HadronConeExclExtendedTruthLabelID=-1, HadronConeExclTruthLabelID=-1, PartonTruthLabelID=-1;
        if(jet->isAvailable<int>("HadronConeExclExtendedTruthLabelID")) HadronConeExclExtendedTruthLabelID = (int) jet->auxdata<int>("HadronConeExclExtendedTruthLabelID");
        if(jet->isAvailable<int>("HadronConeExclTruthLabelID")) HadronConeExclTruthLabelID = (int) jet->auxdata<int>("HadronConeExclTruthLabelID");
        if(jet->isAvailable<int>("PartonTruthLabelID")) PartonTruthLabelID = (int) jet->auxdata<int>("PartonTruthLabelID");
        m_pltree_jet_HadronConeExclExtendedTruthLabelID.push_back(HadronConeExclExtendedTruthLabelID);
        m_pltree_jet_HadronConeExclTruthLabelID.push_back(HadronConeExclTruthLabelID);
        m_pltree_jet_PartonTruthLabelID.push_back(PartonTruthLabelID);
    }
    top::EventSaverFlatNtuple::saveParticleLevelEvent(pEvent);
} //end of saveParticleLevelEvent
```

Run as usual and check the output file in root, e.g.:

```
[ ] particleLevel->Scan("jet_pt:jet_HadronConeExclExtendedTruthLabelID:jet_HadronConeExclTruthLabelID:jet_PartonTruthLabelID")
```

### 3.3 Creating a Custom Selector

AnalysisTop provides a lot of nice options for your event selections, but we cannot cover all crazy ideas analyzers have. Luckily, there is a simple way of adding your own selector. In the `howtoextendanalysisistop` package there is a (silly) example: a selector which allows you to only store events with an even run number for the selections where this is used. Let's try to use it: open `HowtoExtendAnalysisTop/EvenEventNumberSelector.h` and `Root/EvenEventNumberSelector.cxx` and have a



look. The header is very simple, and the core part is:

```
class EvenEventNumberSelector : public top::EventSelectorBase {
public:
    //This function sees every event. If you return true then the event passes this "cut"
    bool apply(const top::Event& event) const override;

    //For humans, something short and catchy
    std::string name() const override;
};
```

i.e. just a declaration of our class inheriting from the base EventSelector class. Now the .cxx

```
//If the event number divided by two has no remainder then return true.
bool EvenEventNumberSelector::apply(const top::Event& event) const {
    return event.m_info->eventNumber() % 2 == 0;
}

//For the cutflow and terminal output
std::string EvenEventNumberSelector::name() const {
    return "EVEN";
}
```

just two notable things: in the `name` method we declare the name we will use for this selector in the config file, and the `apply` method is the ones that is doing the actual selection, in this case returning `true` if the event number is even, `false` if it is odd. Of course in other selectors we could do more complicated stuff (e.g. kinematics cuts on objects we retrieve from the `top::Event` object...).

Ok, let's try now to define our own selector: one that saves all events with an odd number! In the folder with the headers, just copy `EvenEventNumberSelector.h` to `OddEventNumberSelector.h`, and do a similar thing in the `Root/` folder for the .cxx, copying `EvenEventNumberSelector.cxx` to `OddEventNumberSelector.cxx`. Now open the two new files and change all the `EvenEvent` to `OddEvent`. Do not forget to change the "ifndef ... define ..." instruction in the header updating the name, or your code will not be compiled correctly, do not forget to change the header in the .cxx, and in the `name()` method change the return value to `return "ODD";`.

Now in the `Root/` directory open `HowtoExtendAnalysisTopLoader.cxx`, which is the class loading the selectors, and modify it like this:

```
#include "HowtoExtendAnalysisTop/HowtoExtendAnalysisTopLoader.h"

#include "HowtoExtendAnalysisTop/EvenEventNumberSelector.h"
#include "HowtoExtendAnalysisTop/OddEventNumberSelector.h"
#include "TopConfiguration/TopConfig.h"

#include <iostream>
#include "TFile.h"

namespace top{
    top::EventSelectorBase* HowtoExtendAnalysisTopLoader::initTool(const std::string& /*name*/, const std::string& line, TFile* /*outputFile*/, std::sha
    {
        //get the first bit of the string and store it in toolname
        std::istringstream iss(line);
        std::string toolname;
        getline(iss, toolname, ' ');

        //any parameters?
        std::string param;
        if (line.size() > toolname.size())
            param = line.substr(toolname.size() + 1);

        if (toolname == "EVEN")
            return new top::EvenEventNumberSelector();
        else if (toolname == "ODD")
            return new top::OddEventNumberSelector();
        //else if (toolname.find("OTHER_TOOL") == 0)
        // return OtherToolThatYouInvented()

        return nullptr;
    }
}
```

and do a full compilation (because we added new files). Now let's use the new selector: open the `custom-saver-test.txt` config file, and let's add two new selections after our usual `ejets_DL1r_2016`:

```

SELECTION ejets_DL1r_2016
. EL_2016
. ejets_basic
JET_N_BTAG DL1r:FixedCutBEff_77 >= 1
SAVE

SELECTION ejets_DL1r_2016_even
. EL_2016
. ejets_basic
JET_N_BTAG DL1r:FixedCutBEff_77 >= 1
EVEN
SAVE

SELECTION ejets_DL1r_2016_odd
. EL_2016
. ejets_basic
JET_N_BTAG DL1r:FixedCutBEff_77 >= 1
ODD
SAVE

```

You can see we're using the custom selectors there. Run as usual and open in root the resulting file. check it e.g. with

```
[ ] nominal->Scan("eventNumber:ejets_DL1r_2016:ejets_DL1r_2016_even:ejets_DL1r_2016_odd","ejets_DL1r_2016>0")
```

i.e. by printing the event number, and the results for the 3 selections, but only for events passing the `ejets_DL1r_2016>0` condition. Is this behaving as expected?

## 4 Launching jobs on the grid

Ok, we did quite a lot locally. But at some point you will want something more than just running on few events: you'll want to run on millions (billions) of events included in full data/MC samples. The number and sizes of files you will need to run over for a full analysis is very large, so you will need to be able to run on the grid. Since the grid job will take a while to start and run, we will submit these jobs over night. As before, we've put together some handy scripts. They are located in the [TopExamples](#) package.

### 4.1 Retrieving the TopExamples grid scripts

Start with a fresh shell. Then

```

cd ~/work/MyAnalysis//run #or whatever the path is
source setup.sh #if you're using the standard setup file we suggested you, you should uncomment the last lines, which will setup rucio, panda, pyami and voms
mkdir grid
cd grid
cp $AnalysisBase_DIR/bin/01SubmitToGrid_Tutorial.py .
cp $AnalysisBase_DIR/bin/Data_rel21.py .
cp $AnalysisBase_DIR/bin/MC16_TOPQ1.py .
cp $AnalysisBase_DIR/bin/DerivationTags.py .
cd ../

```

### 4.2 Launching jobs on the grid

To submit the grid jobs first need to do some setup (if you didn't do that already as suggested just above):

```

lsetup rucio panda pyami
voms-proxy-init -voms atlas # enter your grid password when you are asked for it

```

Then, the script to launch would be `grid/01SubmitToGrid_Tutorial.py`. Open this file with your text editor. Change some of the settings inside: `settingsFile` is the config file you want to use. You could e.g. want to use

```
config.settingsFile = 'custom-saver-test.txt'
```

or any config file you prefer. There are **at least** two important checks to do **every time** before submitting:

- am I setting `NEvents` to something reasonable for running on the grid? Maybe I wanted to run on 500 events for a quick local test, but on the grid I want to run on the full statistics...
- am I running on all the Systematics or just on the nominal? For this first test, run only the nominal: running all systematics will take MUCH MORE time and MUCH MORE disk space, one should do that only when really needed.

In general, sending jobs on the grid is quite time consuming, so always double-check what you're doing before doing that. Set the argument `config.gridUsername` to your username, and the argument of `config.suffix` to something reasonable (e.g. "firstTest-DATE"): this is a suffix that will help you bookkeeping the datasets you produce, something that will soon become a nightmare if you're not careful. Add also the options:

```

config.maxNFilesPerJob = '10'
config.otherOptions    = '--nFiles=100'

```

In this way we'll run just on 100 files, with a maximum of 10 per job, so we don't have to wait too long for the grid job to be completed. For the time being set `config.noSubmit` to `True`, this is a useful way to test if everything is ok before doing the actual submission.

Below you'll see

```
names = [
    'TOPQ1_ttbar_PowPy8',
    # 'Data16_TOPQ1',
]
```

this is the list of sample lists you want to run on. They're defined in the `Data_rel21.py` and `MC16_TOPQ1.py` and of course you will have to check these, because you want to make sure you're running on something reasonable. Open the `MC16_TOPQ1.py`: you'll see there

```
TopExamples.grid.Add("TOPQ1_ttbar_PowPy8").datasets = [
    'mc16_13TeV.410501.PowhegPythia8EvtGen_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e5458_s3126_r9364_r9315_p3215',
]
```

Ok, so when in `01SubmitToGrid_Tutorial.py` we say we want to run on the "TOPQ1\_ttbar\_PowPy8" list, this is just one sample. Is it reasonable to use this one?

Well, first simple check:

```
rucio list-dids mc16_13TeV.410501.PowhegPythia8EvtGen_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e5458_s3126_r9364_r9315_p3215
```

Ok, it exists! Second quick check:

```
rucio list-files mc16_13TeV.410501.PowhegPythia8EvtGen_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e5458_s3126_r9364_r9315_p3215
```

what? this is empty? It's normal, this is a very old derivation. We learnt at the beginning of the tutorial that from the "r9364" tag we can understand that this is a mc16a sample. Now we have to learn that the `p`-tag is a number which tells you which version of the derivation framework has been used to produce this sample. The lower the number, the older the derivation (usually). Now if you recall, in the first part of the tutorial we used a "p4031" sample (check `input-mc-rel21_p4031.txt` if you don't believe it), so something which is much more recent. How do we know what we should run on?

First we have to choose the correct dataset to represent our physics process. We are interested in running a ttbar l+jets selection in this tutorial, so let's have a look at the [https://wiki.cern.ch/wiki/bin/view/AtlasProtected/PmgTopProcesses#MC16\\_Bulk\\_Sample\\_Status](https://wiki.cern.ch/wiki/bin/view/AtlasProtected/PmgTopProcesses#MC16_Bulk_Sample_Status). Ok, from there we understand that the recommended nominal sample for ttbar l+jets is 410470 with the e6337 tag (the `e`-tag identifies the software version used for the production of the sample at truth level). How do we find the complete name of the dataset? There are several ways, you could for example play a bit with rucio, e.g.

```
rucio list-dids "mc16_13TeV.410470*.DA0D*TOPQ1*e6337*r9364*" | grep CONTAINER
```

- putting the campaign in front is mandatory, i.e. "mc16\_13TeV.", this defines the scope rucio will look at that, otherwise you get an error
- then we specify the derivation format we want, to avoid finding too much stuff. The [TopDerivations](#) are described in their twiki.
- we specify the correct `e`-tag to avoid seeing too much weird stuff
- we specify the `r`-tag because for the first test we just want to run on the mc16a sample

Finally, we only look at containers to avoid looking at single elements of the rucio containers (again, a lot of stuff). If you try this, you will still see a lot of output. Some of the datasets have 2 `r`-tags, like "r9364\_r9315": if possible these should be ignored. Then you will see that some datasets have an "s-tag" (e.g. s3126) and some an "a-tag" (e.g. a875): the first ones are datasets reconstructed with a full simulation, the second ones with a fast simulation. If both are there, and if you don't have a good reason not to, let's use the full simulation (but some datasets are only reconstructed with the fast simulation, to save ATLAS resources). You can find more information on the `a,s,r` tags in the [AtlasProductionGroup](#) twiki page, e.g. [AtlasProductionGroupMC16a](#) for MC16a and similar for the other campaigns. If you want to

so, if we choose a fully simulated sample with a single `r`-tag, we end up with several possible `p`-tags. Can we just live with taking the most recent one? Often yes, but it's always good to double check. In this case we end up with

`mc16_13TeV.410470.PhPy8EG_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e6337_s3126_r9364_p4031`, are we sure this `p`-tag is ok and this is not a buggy dataset? A `rucio list-files` on that dataset tells me that the dataset is filled (and with a lot of events). If I look at the [twiki page for the derivation production](#) (and I open the table there) I can search for the "p4031" tag, and see that that one corresponds to the [AthDerivation](#) release 21.2.84.0 (the releases for the derivations are different and use a different numbering scheme wrt the AnalysisBase ones), and I can see if there are comments. In this case this is marked as "Buggy for LCTopo trimmed jets", but we're not using them, so, (most likely) this should be ok. I can also see that the p4031 tag is listed as a MC tag (data samples use different tags) and that is under the "no-skim" columns: this means that ALL events in the mc sample are stored in the output derivation, not only the ones passing the selection of the derivation (again, look at the [TopDerivations](#) twiki to check the selection applied by a given TOPQX derivation format). In the top group we usually avoid skimming "signal" samples (e.g. ttbar, single-top) but we skim (i.e. we apply the derivation selection when producing the derivation) "background" samples, e.g. V+jets ones. Data derivations are always skimmed.

There are very important resources to keep in mind when looking for a sample in the top group:

- [this page](#) list all the derivations for samples produced with TOPX derivations. Try searching 410470 there!
- [The data derivation lists](#) instead has all the relevant information for the data

To get some more information on the datasets, there are also other important tools to keep in mind:

- [the AMI website](#): try searching for `mc16_13TeV.410470.PhPy8EG_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e6337_s3126_r9364_p4031` there, you can find a lot of information, find the provenance of the dataset, etc...
- using [the AMI tag browser](#) you can get information on what a particular tag means: try searching "p4031" there! There is also a very useful "compare" function if you want to understand the difference between two datasets differing only by a tag
- finally, you can use from command line (on lxplus e.g.) `pyami`, instead of using the website: try `ami show dataset info mc16_13TeV.410470.PhPy8EG_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e6337_s3126_r9364_p4031` and check the output. [Some documentation is here](#)

Ok, enough. We want to run on the grid on `mc16_13TeV.410470.PhPy8EG_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e6337_s3126_r9364_p4031`. In `MC16_TOPQ1.py` change the "TOPQ1\_ttbar\_PowPy8" to

```
TopExamples.grid.Add("TOPQ1_ttbar_PowPy8_mc16a").datasets = [
    'mc16_13TeV.410470.PhPy8EG_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e6337_s3126_r9364_p4031',
]
```

in `01SubmitToGrid_Tutorial.py` change the lists we want to run on:

```
names = [
    'TOPQ1_ttbar_PowPy8_mc16a',
    # 'Data16_TOPQ1',
]
```

Finally, try to run `python grid/01SubmitToGrid_Tutorial.py`

You should see that the python scripts are just putting together a command line like

```
prun \
--inDS=mc16_13TeV.410470.PhPy8EG_A14_ttbar_hdamp258p75_nonallhad.deriv.DA0D_TOPQ1.e6337_s3126_r9364_p4031 \
--outDS=user.mvanadia.410470.PhPy8EG.DA0D_TOPQ1.e6337_s3126_r9364_p4031.test-17-04-2020 \
--useAthenaPackages --cmtConfig=x86_64-centos7-gcc8-opt \
--writeInputToTxt=IN:in.txt \
--outputs=output_root:output.root \
--exec="top-xaod custom-saver-test.in.txt" \
--outTarBall=top-xaod.tar.gz \
--noSubmit \
--mergeOutput \
--memory=2000 \
```

From outDS you can see the name that the output dataset will have (the dataset for the root files will be e.g. in this example above called "user.mvanadia.410470.PhPy8EG.DA0D\_TOPQ1.e6337\_s3126\_r9364\_p4031.test-17-04-2020\_output\_root/", in your case username and suffix will be different). You can refer to the [prun documentation](#) for more information on prun. Now try to set `noSubmit False` (after making sure everything is really ok) and submit for real.

## 4.3 baby-sitting the jobs

After few minutes, you should see the job appearing [on your PANDA monitor](#). You can check there the progress of your job, retrieve the logfiles in case of errors (click on the job ID Parent -> Show jobs -> All -> click on the PANDA ID of one of the subjobs -> Logs -> Lof files) and do a lot of useful stuff.

Another tool to baby-sit jobs is [pbook](#).

```
lsetup panda
pbook
```

To get full help, run `help()` command. Some basic features include commands, such as

```
retry(taskID) # retry a task in failed state, or retry failed jobs of a still-running task
kill(taskID), kill('running') # will kill all running tasks
kill(taskID1, taskID2) # will kill all tasks with ID >= taskID1 and ID <= taskID2
```

When at least part of the subjobs are completed, try to download at least some files you produced and have a look at them. You can do that in your workarea on lxplus if you have the space, but you can for example go to your eos area on lxplus (=eos/user// where you have 2 TB of space available), and (after setting up rucio) do:

```
rucio download --nrandom 5 user.USERNAME.410470.PhPy8EG.DA0D_TOPQ1.e6337_s3126_r9364_p4031.SUFFIX
```

You should get 5 files among those you produced on the grid. Another place where you can find some disk space is on the lxplus scratch disks (= cd /tmp/ =), but be aware that files are kept only for a very limited time there, then they are deleted. Otherwise, you should use resources granted from your institution.

## 4.4 I have a sample to run on but there is no TOPQX derivation available

Refer to the [TopDerivations](#) twiki.

## 4.5 other useful features

While you wait some hours/days for your job to finish, you can familiarize with some other useful options in the submit script

```
config.maxNFilesPerJob = '10' # each job in a pandatask will process no more than 10 input files
```

It's not recommended to select a maximum number of input files per subjob, because usually this will just make your code running slower on the grid, but in some cases it can be useful to avoid subjobs being too long.

```
config.reuseTarBall = False # compress the files for grid submission once and reuse the tarball for all submissions (NOTE: will ignore any changes to
```

You may notice in the folder where you submitted your files a new file, `top-xaod.tar.gz`. This is a tarball including all things you sent on the grid. Now, if you switch this to True, every time you recall the script you'll reuse the same tarball. This is quicker **but dangerous**, because if you change any file locally but you don't redo the tarball then you'll not take the changes. So only use this option if you're sure of what you're doing.

```
config.excludedSites = ''
```

In some cases some grid sites become problematic; you can avoid sending jobs there with this command (until the problem is fixed).

```
config.destSE = ''
```

This command is to automatically store the output of your job on some grid site: for example, your Institution may have some localgroupdisks on the grid, and if you have the proper permissions (contact your local IT department for that) you can store directly the samples you produce there, and this is much faster than having to manually downloading them with rucio. However, we do not advice doing this replica with this option, but it's rather more convenient to use [the rucio web interface](#).

(Data Transfer -> Request new rule). You can request a replica to a local group disk of the dataset you produced from few minutes after sending the production on the grid, even if this is not complete, and you can easily check the size of what you're trying to copy (ensuring you do not overfill your quota) and the lifetime of the dataset. You will quickly realize disk space is your biggest enemy in analysis, so doing things in a smart, controlled way is not recommended, it's mandatory! There are some common resources for the top group, as described [in this twiki](#), but they are EXTREMELY limited and can be used only in some cases.

## 4.6 Taming grid issues

The information above is a baseline that should be a starting point for investigating issues with jobs. We will try to expand on some typical issues when running on grid below. Note that this section is more like a short reference rather than an interactive tutorial (and yes, we are not kidding about the briefness of this section when considering how much stuff can go wrong on grid...)

**I changed my code/scripts, submitted another task, but my changes are not reflected at grid!** Check if you have `config.reuseTarBall=True` option in the submission script. If yes, this means that if a compressed tarball that is sent to grid exists in the directory (`top-xaod.tar.gz`) **it has been re-uploaded without actually updating with your changes!** This option is very nice when submitting many jobs at once, but if you use it, make sure to always delete the `top-xaod.tar.gz` tarball before you submit new tasks with changes!

**Jobs take forever to start:** This can be seen by large time to start of the job. This could be related to your priority (check the priority of the jobs running on the site). If this is not the case, the site may be busy (too many jobs in the site queues) or some other issue (in which case it may be advisable to ask the [DAST experts](#)).

**Jobs crash:** In this case the error message displayed in the job list may or may not be helpful, in any case it is useful to examine the log files. Log files can be accessed by opening the specific crashing job and examining log files ([log files on panda](#)), in particular the `athena_stdout.txt` log that includes output from AnalysisTop run (or any other task you run on grid in general).

**Jobs take forever to run or crash due to exceeding site walltime limit:** Check if the job is running long for the first time, or if it has been retried. If the latter, this may indicate some issue with the code (a crash perhaps, see point above), or some issue related to the grid site potentially. Note that all grid sites have a limit on maximum time of a job (walltime). Thus you may have an issue due to your job being simply too long -- either the time to process an event is on average very high and/or the number of events to process in a single job is very high. This can become an issue in particular, if you are processing many systematics and/or using CPU-intensive algorithms, such as complex kinematic reconstruction algorithms (e.g. KLFilter). In this case, lowering the number of files per job may be desired. When submitting a sample to grid, this can be achieved by the `config.maxNFilesPerJob` setting, limiting the maximum amount of files per job. Alternatively, if you are retrying a job via pbook, it is possible to use the `nFilesPerJob` option specifying exact number of files per job (note, that you cannot use `maxNFilesPerJob` in pbook):

```
retry(taskID, newOpts={'nFilesPerJob':5})
```

Ideally, it is useful to download a single file from the offending sample and try to run it locally to get an idea how long does it take to process N events and adjust the number of files accordingly to fit into the site walltime. For full-systematics NTuple production with KLFilter for instance, it is not unexpected having to run with single file per job for signal samples (and any samples that have high event selection efficiency in general)!

**Broken tasks:** Each task runs several (typically 10?) scout jobs, which are used to check if the task does not crash, if the site meets requirements such as sufficient memory, etc. If all of the scout jobs fail, the task is sent to broken state, and no further jobs of the task are executed. This calls for investigating why the jobs crashed and fixing the issue. Either your jobs have crashed due to your fault, or the crashes are site-related, in both cases you encounter an issue, where the job to be resubmitted needs a unique name. It is possible to resubmit a job with identical name, by running the `01SubmitToGrid.py` script with the specific sample and additional parameter:

```
config.newOpts = ' --allowTaskDuplication'
```

BEWARE: When downloading output from grid, output of both the broken and the resubmitted task will be downloaded. In case of a broken task where all jobs crashed, there is no output, so in such a case it is safe to use this option. Additionally, in case of site-related issues, it may be necessary to exclude it using the `config.excludedSites` option.

**Retrying failed task in pbook on a different site:** Sometimes, the only way to overcome a problem is to retry a job on a different site. This can be done using pbook:

```
#to exclude sites XXX and YYY, where XXX and YYY must match the bigpanda name, i.e. without the "ANALY" prefix
retry(taskID, newOpts={'excludedSite':'XXX,YYY'})
```

**Debugging memory-related problems on bigpanda:** Majority of the grid sites have a limit on memory for single-core jobs of 2GB. In general if your job cannot fit into 2GB single-core, you have a problem. In case of memory leaks, typically panda will show warnings on the task details page about jobs consuming excessive amount of memory, or kill your jobs right away. This is illustrated in our already shown [example task joblist](#) (we ran a job with intentional large memory leak, that consumed 4GB of RAM in a matter of minutes). It is possible to view the memory consumption and input/output (I/O) of job as a function of time in graphs in the [detailed view of a specific job](#) and opening the [Memory and IO plots](#).

**Large RAM consumption due to branch buffer optimization:** Another frequent issue in case of large NTuples (many systematics and or plenty of branches) is the RAM consumption of the branch buffer optimization. This problem manifests itself by a sharp increase in RAM consumption after a certain amount of time, when output events were processed which are about to be written to file. By default TTrees cache in memory and the output is written out every 1000 events. An example of memory and I/O behavior indicating this issue can be seen in the following [memory and I/O plot](#) from an [example task](#), with full systematics and a large vector variable stored in each systematic TTree. The easiest recommendation is to reduce the size of NTuples as much as possible and consider if some of the stored variables could be computed offline instead. Another option, if running with systematics, is to split systematics into multiple jobs. Finally, if that is not enough, it is possible to play with the following parameter in AT configuration file, **but please mind that THIS MAY LEAD TO I/O PERFORMANCE ISSUES** when reading the produced NTuples:

```
OutputFileBasketSizePrimitive 4096 # initial buffer size for bool, int, float, ...
OutputFileBasketSizeVector 40960 # initial buffer size for vector
OutputFileNEventAutoFlush 1000 # number of saved events after which buffer sizes are optimized
```

More realistic values for initial buffer sizes may reduce the amount of memory needed for the reoptimization and decreasing the `OutputFileNEventAutoFlush` value reduces the amount of cached events to be written out, but **note that lowering the value does also reduce the I/O performance for the output files (meaning, offline analysis of these files will have slower read performance as well)**.

## 4.7 Monitoring dataset obsolescence

Derivations and also AOD samples used to produce the derivations are cancelled from time to time, because we are always very limited by the disk space in ATLAS. Information on how to avoid losing important data is [here](#), and this topic will also be covered in the [Advanced tutorial](#).

## 5 Normalisation to the integrated luminosity

In most if not all physics analyses, you have to compare what you observe in a sample of real data with the prediction from one or several simulated samples. To do so, the MC simulated samples have to be normalised according to the integrated luminosity of the data sample. This is done by applying the following weight to each MC event:

$$w = \frac{\mathcal{L} \cdot \sigma}{N_{MCevents}}$$

This weight is so calculated that, when processing  $N_{MCevents}$  simulated events without applying any selection, the yield will be equal to the product of the cross-section  $\sigma$  of the simulated physical process by the integrated luminosity  $\mathcal{L}$ . Since you need to know how many events you have processed for both data and MC, this weight need to be computed after you have produced your Ntuples with AnalysisTop, possibly on grid.

### 5.1 Retrieving the integrated luminosity

Currently, AnalysisTop doesn't have a functionality to retrieve the integrated luminosity corresponding to the data samples you processed. Therefore, you retrieve this information by yourself. Once the data samples are made available, this information is made available in the usual top-reco twikis, currently [TopDerivationMC16aList](#).

### 5.2 Retrieving the number of processed MC events

In the output files of AnalysisTop, the number of processed MC events is stored in a separate tree: `sumWeights`. Each entry of this tree corresponds to one input file. It should contain three branches:

```
root -l output.root
root [] sumWeights->GetListOfLeaves()->Print()
Collection name='TObjArray', class='TObjArray', size=16
OBJ: TLeafI dsid dsid
OBJ: TLeafF totalEventsWeighted totalEventsWeighted
OBJ: TLeafL totalEvents totalEvents
```

The `dsid` gives you the dataset number (DSID) of the MC sample from which the file was created. This will allow you to retrieve the corresponding cross-section (see below). The DSID should be identical across all input files / tree entries (unless you run very weird setups). When coding for your analysis, keep in mind that the DSID does not uniquely identify a sample, e.g. a full-simulation and an AtlFast sample might have the same DSID.

The `totalEventsWeighted` is a float which gives you, for each of the processed input file, the weighted number of events without any selection: this is what you should use to calculate  $N_{MCevents}$ , by looping on the entries of the `sumWeights` tree.

"Weighted" means that the MC generator event weights have been applied to compute this sum. Note that these weights can be very different from 1 depending on the MC generator; for NLO generators the weight can be negative, and for Powheg-Box the weights can be very small. Therefore, `totalEventsWeighted` may not be in the general case an integer. This is normal; as long as you have applied this MC generator weight to each event (it is stored as `mc_weight` in the nominal tree as in the trees corresponding to systematics), everything should be ok. Note that if you perform truth-level reweightings (e.g. PDF reweighting), you'll have to use a number that's different from `totalEventsWeighted`.

### 5.3 Retrieving the cross-section of the MC sample

The generator cross-sections and K-factors (to be multiplied together to get the full cross section) for all the MC samples used in ATLAS can be retrieved in the following twiki page maintained by the ATLAS Physics Modelling Group: [CentralMC15ProductionList](#). However, the Top working group maintains their own list of cross sections in the [sample-xsections](#) repository. The file [XSection-MC16-13TeV.data](#) in particular contains the cross-sections of the MC samples (among other things).

One convenient way to retrieve the cross-section of a given DSID is to use the `TopDataPreparation` package from within your AnalysisTop session. You need to create a `SampleXsectionSvc`, pointing it to the correct text file containing the cross-sections:

```
root [] SampleXsectionSvc* s = SampleXsectionSvc::svc("XSection-MC16-13TeV.data")
```

If you don't give the full path of the text file, it will be searched for in the [TopDataPreparation](#) /data directory, which may be severely outdated. Hence either download the [sample-xsections](#) repository, or use the file directly from CVMFS:

```
/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/TopDataPreparation/XSection-MC16-13TeV.data
```

The `SampleXsectionSvc` object is a singleton (only one instance can exist) which holds a const `SampleXsection` object which is initialised automatically with the text file, and does the mapping of the MC samples to their cross-sections. You can retrieve the cross-section of the MC sample corresponding to DSID 410000 by doing either of these:

```
root [] s->sampleXsection()->getXsection(410000)
(double) 4.516641e+02
root [] (s->sampleXsection()->getRawXsection(410000))*(s->sampleXsection()->getKfactor(410000))
(double) 4.516641e+02
```

If you prefer doing things in Python, you may get some inspiration from the `checkSyntax` function in the [checkSyntax.py](#) script.

## 6 Internal generator reweighting

An important feature you may need to estimate some systematics is the internal generator reweighting. In most modern samples, event weights are computed at



the generation stage, which when applied instead of the nominal `weight_mc` give the same MC sample as if it was generated with a different generator settings. These generator settings can be factorisation or renormalisation scales, different PDFs, etc.

In order to enable this feature, you need to use the `MCGeneratorWeights` option in your cut-file. If you set this option to `True`, a new branch called `mc_generator_weights` would be added to the `truth` tree for all processed events. If you want to get this branch only for those which pass your `r`(reco-level) selection cuts, you should put this option to `Nominal` instead, and the branch would be added to the `nominal` tree instead.

This `mc_generator_weights` branch is a vector of weights, with the same size for each event, corresponding to the number of weights which were added when preparing the MC sample. The first entry in the vector should be equal to the nominal `weight_mc` which you apply to each MC event by default. Then, if you replace this `weight_mc` by any of the other entries you would get a reweighted sample, for e.g. a different PDF set.

But to normalise you sample to the integrated luminosity - as explained in the previous section - you need to update consistently the sum of weights  $N_{MCevents}$ . For this you should use the new branch named `totalEventsWeighted_mc_generator_weights` in the `sumWeights` tree, in place of the `totalEventsWeighted`. The `totalEventsWeighted_mc_generator_weights` is a vector, each entry `i` of this vector contains the sum of the weights of entry `i` of the `mc_generator_weights` vector. NB: in the most recent derivations, this sum of generator weights is extracted from the meta-data, as is the `totalEventsWeighted`. In older samples where we don't have this information stored, AnalysisTop would recompute this sum on-the-fly; in such case you should be sure to run on all generated events (e.g. it wouldn't work if you are using skimmed derivations).

Then of course you need to know which new weight you need to use. For this there is the `names_mc_generator_weights` in the `sumWeights` tree; this vector of strings will give you the names of these weights. However, when running on older samples where this info wasn't stored in the meta-data, you would just get a `?`. In that case you need to run an external tool, such as the `check_MetaSG.py` script (but you probably won't need it for MC16 samples).

## 7 Some more details on PRW

We already mentioned the three MC16 campaigns and the different settings that have to be used for the pileup reweighting for them. We also mentioned that the reweighting is based on the average number of collisions for mc16a, and on the actual number of collisions for mc16d and mc16e. Details on what this means and on the PRW are on the [ExtendedPileupReweighting](#) twiki page. The settings to use are reported here (included from [this twiki](#)) :

If you are running AnalysisTop code, the you should use following lines for the Lumicalc, GRL and PRW files. **\*Note: this applies only when you are using the TOROID-ON dataset\***

```
GRLDir GoodRunsLists
GRLFile data15_13TeV/20170619/physics_25ns_21.0.19.xml data16_13TeV/20180129/physics_25ns_21.0.19.xml
PRWConfigFiles_FS dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16a.FS.v2/prw.merged.root
PRWConfigFiles_AF dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16a.AF.v2/prw.merged.root
PRWLumiCalcFiles GoodRunsLists/data15_13TeV/20170619/PHYS_StandardGRL_All_Good_25ns_276262-284484_oflLumi-13TeV-008.root GoodRunsLists/data16_13TeV/20180129/PHYS_StandardGRL_All_Good_25ns_276262-284484_oflLumi-13TeV-008.root
```

For 2017 (mc16d) it is **strongly recommended** to use actual mu reweighting:

```
GRLDir GoodRunsLists
GRLFile data17_13TeV/20180619/physics_25ns_TriggerNo17e33prim.xml
PRWConfigFiles_FS dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16d.FS.v2/prw.merged.root
PRWConfigFiles_AF dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16d.AF.v2/prw.merged.root
PRWActualMu_FS GoodRunsLists/data17_13TeV/20180619/physics_25ns_TriggerNo17e33prim.actualMu.OflLumi-13TeV-010.root
PRWActualMu_AF GoodRunsLists/data17_13TeV/20180619/physics_25ns_TriggerNo17e33prim.actualMu.OflLumi-13TeV-010.root
PRWLumiCalcFiles GoodRunsLists/data17_13TeV/20180619/physics_25ns_TriggerNo17e33prim.lumicalc.OflLumi-13TeV-010.root
```

and the same is true for 2018 (mc16e):

```
GRLDir GoodRunsLists
GRLFile data18_13TeV/20190219/physics_25ns_TriggerNo17e33prim.xml
PRWConfigFiles_FS dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16e.FS.v2/prw.merged.root
PRWConfigFiles_AF dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16e.AF.v2/prw.merged.root
PRWActualMu_FS GoodRunsLists/data18_13TeV/20190318/physics_25ns_TriggerNo17e33prim.actualMu.OflLumi-13TeV-010.root
PRWActualMu_AF GoodRunsLists/data18_13TeV/20190318/physics_25ns_TriggerNo17e33prim.actualMu.OflLumi-13TeV-010.root
PRWLumiCalcFiles GoodRunsLists/data18_13TeV/20190318/ilumicalc_histograms_None_348885-364292_oflLumi-13TeV-010.root
```

The corresponding luminosity can be found in the table below. Up-to-date information on luminosities and GRL and lumicalc files can be found on this [twiki](#).

Year	Nominal Luminosity value (fb <sup>-1</sup> )
2015	3.220
2016	32.99
2017	44.31
2018	58.45

### 7.1 Inspecting the PRW files

A common PRW file that contains most of the relevant files for top physics can be found here:

```
/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/PileupReweighting
```

These files are updated automatically once a TOPQ derivation for a given dataset is produced. When a new MC samples is produced, automatically NTUP\_PILEUP files are produced. These are the files needed for the pileup reweighting. More information about the common files can be found in [TopPileupReweightingFilesMC16](#).

**Note** That PRW files for full simulation (FS) and fast simulation (AFII) as these files were generated with a different seed.

You can open one of the PRW files using `TBrowser` and you can inspect the pile-up distributions for the different DSIDs

```
root -l /cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/PileupReweighting/user.iconnell.Top.PRW.MC16d.AF.v2/prw.merged.root
root [1] TBrowser b
```

## 7.2 Generating my own PRW file

It is possible to prepare your custom PRW file. All you need to do is to download all the relevant `NTUP_PILEUP` files (these should be produced automatically with the MC production) and then simply merge them using the `hadd` command. Then you just need to point to this file in your config file

```
PRWConfigFiles_FS path/to/my/custom/prw.root
PRWConfigFiles_AF path/to/my/custom/prw.root
```

**Warning** Do not forget to include your custom PRW file in the tarball that you send to grid via `--extFile path/to/prw.root` in the `prun` command.

## 7.3 Using actual mu reweighting

The actual mu reweighting is reweighting based on "the mean number of minbias collisions" while the average mu (the default) "is averaging the 'actual mu' over the colliding bunches". **It is strongly recommended to use actual mu reweighting for mc16d and mc16e, while this option is not available for mc16a.** In practice, both of the techniques should reproduce the pile-up conditions in data well. The difference is that using the average mu reweighting sometimes produces weights far from 1 thus effectively reducing the statistical precision of the MC sample. So if the MC statistics is a concern, you should use the actual mu reweighting, this is estimated to enhance the statistical power by up to 20%. More information is provided in [ExtendedPileupReweighting](#).

To use the actual mu reweighting, you need to add these two lines to the config file for mc16d

```
PRWActualMu_FS GoodRunsLists/data17_13TeV/20180619/physics_25ns_TriggerNo17e33prim.actualMu.0fLLumi-13TeV-010.root
PRWActualMu_AF GoodRunsLists/data17_13TeV/20180619/physics_25ns_TriggerNo17e33prim.actualMu.0fLLumi-13TeV-010.root
```

**Note** actual mu reweighting is available only for mc16d and mc16e campaigns and is not available for mc16a.

Which variables should you use to verify that the reweighting works correctly? When you use actual mu reweighting you should look in the output trees at the variable called `mu` which is in fact average mu. So standard Data vs MC plots for this variable should show you how good the agreement is. When you use actual mu reweighting you should look at variable called `mu_actual`. Any other combination is not expected to give you good data vs MC agreement.

**Note** In MC, the `mu` and `mu_actual` will be the same (it has to do with how the MC is produced), however it will be different in data!

## 8 More on b-tagging and CP recommendations

All physics objects are calibrated in AnalysisTop using software tools provided by the CP groups. In the AT code, the CP tools are configured in [this package](#) (no need to browse its code right now) and then used in several other parts of the code to apply calibrations and calculate Scale Factors. By choice, we don't give full access to the users to the CP tools, i.e. they cannot directly modify their settings, but some of the settings used can be modified using the config options (see above for the full list). SFs are usually stored by CP groups in so-called CDI files, which are usually root files containing the numbers needed for calculating the SFs. These are kept on a dedicated cvmfs area: `/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/`. In general, the latest version of CDI files should be used, unless differently announced due to temporary problems, or for particular use cases.

Usually CP groups release new versions of the CDI files when they release new recommendations. Users in AT cannot decide which CDI file is used: this is completely set by the AT release used. Advanced users may of course download the AT packages locally with a sparse checkout and modify there the CDI file set in the code (see e.g. [this twiki](#)), but we provide no config option to update the CDI file: this is done on purpose, to ensure most up to date recommendations are used, that recommendations are correctly implemented in the code, and the reproducibility of which recommendations are used with which AT release.

There is one exception: the b-tagging CDI file can be set using the config option `BTagCDIPath`: this is because recent changes in the b-tagging CP tools introduced changes that made not all CDI files usable with all derivations p-tags. Further more, as you saw above, jet collections have now a time-stamp as a suffix: e.g. `AntiKt4EMPFJet_BTagging201903`. More information on this topic from the CP groups are available [on this twiki](#). The important take-home message is that not all derivation caches (i.e. the software version used to produce the derivation sample, identified by its p-tag as described above) are compatible with all jet collections and with all CDI files: it's somehow a 3-dimensional problem. A not-allowed configuration will be easily spotted: it will crash: from the errors it should be clear that the problem is that you're using some combination of p-tag and jet collection which is not usable with a specific CDI file. We'll try to add information on allowed configurations [on this twiki](#).

If you want to manually explore the b-tagging CDI file, to have a look at its structure, you can find them in the folder `/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/xAODBTaggingEfficiency/13TeV/`

For more information on CP recommendations for objects in AnalysisTop, [we have a dedicated twiki](#). Other information on CP recommendation are available on the twiki pages of the CP groups (you'll find them all linked in the homepage of the [Atlas Physics Twiki](#)).

## 9 The TopDataPreparation file: cross-sections and showering algorithms

We already encountered above the `TopDataPreparation` files: these are files containing information on the cross-section, k-factors and showering algorithms for MC samples. The most important one is `XSection-MC16-13TeV.data`, which can be seen: [in the git repository](#) or on cvmfs `/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/TopDataPreparation/XSection-MC16-13TeV.data`

The file format is something like:

```
410000 377.9932 1.1949 pythia
410001 377.9932 1.1949 pythia
410002 377.9932 1.1949 pythia
```

where the first column is the dataset ID number (DSID), the second one is the cross-section times the efficiency of the generator-level filters applied, the third one is the k-factor to rescale the cross section to theoretical predictions at higher orders, and the fourth one is the parton shower generator used, e.g. pythia, pythia8, herwig...

The cross-section information included in this file are often used by analyses for the normalization of their samples (we discussed this topic already in this tutorial),

but they are not really used by the AnalysisTop code when `top-xaod` is executed. The part which is really needed by `top-xaod` is the showering algorithm: this is because the b-tagging CP tools need to know what is the parton shower generator used for showering the sample, since different SFs are needed for b-tagging depending on the algorithm. This is explained in more detail [on the FTAG twiki page](#). Therefore, if you run on a MC sample that's not included in the `XSection-MC16-13TeV.data` file, you will get a crash saying something like `showering algorithm not found` for that sample. If you want to solve this, you can open [a JIRA ticket on the AT board](#) and ask the AT managers to add the sample to the file: this will usually happen in a matter of hours. See e.g. [this example](#). This is by far the recommended option, since AT managers will help you cross-check you're filling correctly the information. If really needed, another way to go is to create a local custom TDP file and use that one instead, using the `TDPPath` option in the config file. More information on the topic [are in the start guide](#).

Before ending this section, we need to discuss some technical details on how these files are provided to AnalysisTop. By default, the `TDPPath` option in the config file is set to be `TDPPath dev/AnalysisTop/TopDataPreparation/XSection-MC16-13TeV.data`, which means that the `/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/TopDataPreparation/XSection-MC16-13TeV.data` file will be used by `top-xaod`. If you have a look at that file e.g. with `ls -ltrh /cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/TopDataPreparation/XSection-MC16-13TeV.data`, you will realize that one is actually a **soft link** to another file inside the `/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/TopDataPreparation/versions/XSection-MC16-13TeV.data/` folder. This is because every time a new DSID is added to the file, the AT managers will then copy that to `cvmfs` and will create a new version of the file in the `versions/XSection-MC16-13TeV.data` folder: each version has a time stamp and a unique code to make that identifiable. The `/cvmfs/atlas.cern.ch/repo/sw/database/GroupData/dev/AnalysisTop/TopDataPreparation/XSection-MC16-13TeV.data` soft link is always pointing to the latest version of the file: this is why this is used by default, because usually this is the right choice. There may be reasons however to not want that, e.g. in cases when the soft link becomes broken (unfortunately it happened lately due problems on eos), or because you want to be sure to use ALWAYS exactly the same TDP file for a given production: you may want to set in the config file a specific version to be used, e.g. with `TDPPath dev/AnalysisTop/TopDataPreparation/versions/XSection-MC16-13TeV.data/XSection-MC16-13TeV-20200420-bd8a7a7c.data`

## 10 Saving disk-space

We already mentioned it: the disk space is often the biggest problem for analyses. If you don't take care of optimizing your ntuples, at some point you will have big problems and delays. There are several ways of reducing the size of your output:

### 10.1 Using DynamicKeys to control my CustomEventSaver

We already discussed above how to use **DynamicKeys**: you could define one to control when you store some extra variables in your custom event saver and when not (maybe you need them only on the signal and not on background samples?)

```
DynamicKeys StoreExtraVariables
StoreExtraVariables True
```

and then you use the same technique we discussed above to get the value of this boolean inside the code and decide if you want or not to store the variables in output trees, for example.

### 10.2 Filtering out branches and/or trees

You can remove branches from the reco-level trees using the `FilterBranches` option in the config file, and do the same at particle-level (parton-level) with `FilterParticleLevelBranches` (`FilterPartonLevelBranches`). Wildcards are allowed. You can remove whole systematic trees with the `FilterTrees` option, and completely remove the particle-level (parton-level) tree using `TopParticleLevel False` (`TopPartonLevel False`), if you're not interested in them

This is explained in [this twiki](#).

Again, **it's very important to do this carefully before sending a massive production on the grid which will be extremely slow and impossible to store on disks**. In particular adding all systematics without thinking is just a HUGE waste of time and resources.

### 10.3 Do not just dump particle/parton level stuff into the reco trees

Maybe you want to do unfolding for your analysis, and sometimes you want to dump some particle-level or parton-level information in your reco-level trees, because having those variables in separate truth-level trees is not convenient. This can work in very simple cases, but will not work for most of the cases. The best thing to do is to just keep what is reco-level in the reco-level trees, and what is particle-level in the particle-level trees: you can always run in a synchronized way on them, as discussed for example [on this twiki](#).

## 11 How to report a problem or request a new feature

This is a very important thing to save your time and the time of people trying to help you. At some point you will have a problem with AnalysisTop. The first thing to do is to make sure the problem is really related to AnalysisTop: for example, problems related to grid behaviour (and not due to AT behaviour on the grid) should be reported to **AtlasDAST** instead. If you're sure this is an AnalysisTop problem, in order:

- make sure your local installation didn't get screwed up (this happens sometimes). You may see weird compilation or even running problems disappearing when removing the `build/` directory, recreating it and recompiling it from scratch. Also, do that starting from a fresh shell
- from the messages you get, try to understand from the error where is the problem coming from, and check carefully your code and config options. Sometimes very weird things (e.g. a character copied&pasted by mistake somewhere, or even an invisible special character introduced somehow somewhere, may lead to problems).
- try using a debugging tool like `gdb`. We'll cover this in our [Advanced tutorial](#) this year!
- check standard top-reco twikis (like the start guide) to see if you're using correctly the feature causing the crash.
- look at the [FAQTopRun2](#) (which we're re-updating now) and the relatively recent announcements on the [hn-atlas-top-reconstruction](#) mailing list ([full archive here](#)) to make sure this is not a known problem.
- finally, either open a JIRA ticket on the [AnalysisTop JIRA board](#) (which is the suggested way), or send an email to [atlas-phys-top-topreco-atmanagers](#) (if you think this is a problem affecting only you) or [hn-atlas-top-reconstruction](#) (in case you think your problem can be a general one)

You should always include the following information:

- version of the release used, config file and input files used, in general anything that can help reproducing your problem

- the error message you get, and possibly a complete log file showing the problem
- any useful input you may have for finding the problem

The more precise the information you send from the start, the quicker the replies will be, and the sooner you will find a solution.

For requesting new features in AnalysisTop (after making sure they're not already there or easily doable with a custom saver for example) ALWAYS open a ticket on the [AnalysisTop JIRA board](#) with a clear explanation of your proposal, why this is interesting in general and not only for your specific case (if it's something very specific, usually the central code is not the right place for that), and possibly with a first proposal on how to implement the change you want. After discussing the new feature with the AT managers and the conveners, they may ask you to implement that by yourself, following the instructions on the [AnalysisTopGuideToUpdatingAthena](#) twiki.

For requesting new features/variables in [TopDerivations](#) refer to the appropriate JIRA board: <https://its.cern.ch/jira/projects/TOPQDERIV/>

## 12 Running multiple configuration files at the same time

### 12.1 First let's do a test locally

Ok, let's say now that you want to test different options for your analysis. For example, you want to test what happens if you use EMTopo jets instead of PFlow jets. We played a lot with config files already, let's start with a fresh one. Setup as usual, then in the `run` folder do:

```
cp $AnalysisBase_DIR/data/TopAnalysis/example-config.txt .
```

Set it to run on 500 events and set the output file name; also turn on the writing of the parton-level tree, we will use this later:

```
NEvents 500
...
OutputFilename output-EMPFlow.root
...
TopPartonLevel True
```

run as usual on the filelist we have `top-xaod example-config.txt input-mc-rel21_p4031.txt`. Now let's prepare a second file to run on topo jets: copy the config file `cp example-config.txt example-config-topo.txt` open the new file and change the jet collection option, and also change the output file name:

```
OutputFilename output-EMTopo.root
...
JetCollectionName AntiKt4EMTopoJets_BTagging201810
```

Let's say now we want to do a quick comparison of these two. Let's take advantage of the example plotting script (`cp $AnalysisBase_DIR/bin/validation.py my-validation.py` if you don't have it already).

In the `my-validation.py` script, make sure you're producing the plots for jets (and not for objects you don't have defined in the config file), and for selections you actually are running:

```
channels = ['ejets_DL1r_2016', 'mujets_DL1r_2016']
particles = ['Electrons', 'Muons', 'MET', 'Jets']
```

```
rm comparison_plots/* # to clean the plots we produce before; create the directory if you don't have it already
python my-validation.py comparison_plots/ output-EMTopo.root output-EMPFlow.root
```

(Hopefully) this worked, and we can check e.g. the difference in leading jet pt between the two selections very quickly, e.g. with

```
display comparison_plots/mujets_DL1r_2016_jet1_pt.png &
```

do you see the difference?

### 12.2 Merging output from different config files

Let's say now that we want to merge the two root files we created in a single one; this can be useful for different reasons, some of which we'll see later. The problem is that both `output-EMTopo.root` and `output-EMPFlow.root` have Trees which have the same name, so with "hadd" they would be just merged in the same TTree, which is something we really don't want. Luckily, there is a useful script for this:

```
combineATOutputs output-combined.root output-EMPFlow.root:EMPflow,output-EMTopo.root:EMTopo
```

The syntax is quite intuitive:

- `output-combined.root` is the name of the output file
- then this needs a comma-separated list elements with the syntax `XXX:YYY`, where `XXX` is the root file you want to merge, and `YYY` is the prefix you want the TTrees in that file to have after the combination.

The output of that command is a file called `output-combined.root`. Open it in root and explore its content: you will see now TTrees like "EMPflow\_nominal" and "EMTopo\_nominal". So now we have a single file with different trees produced with different configurations on the same events: this is quite useful for comparisons! We can actually do quite interesting things with this technique. But before going into that, few rules for `combineATOutputs`:

- you can't specify twice the same prefix or input file
- one of the prefixes can be empty, not more
- the output file should be different from all the input files
- TTrees may be identical - will get identical `Topo_sumWeights` and `Topo_sumWeights`

You can see that also the truth Trees have been merged: why is that? In principle truth is the same, irregardless of what we do at reconstructed level... There are good reasons to have separate trees also at truth level though:

- one can also run config files with different settings at truth level
- one can use truth-level trees, in case when they are supposed to be identical for the different config files we used, to check that we're doing something reasonable and that we're ACTUALLY using the same events

## 12.3 Analysing the combined output: synchronizing different TTrees

ROOT allows running on different trees in parallel in a synchronised way. This feature is extremely useful: a typical use case is the unfolding, where you want to look at the same event at truth level and at reconstructed level, which you usually have in two separate trees, and check e.g. if an event passing the truth-level selection passes also the reco-level one or not. Or, in our example here, maybe we want to check what happens to the [EtMiss](#) when we use EMPflow jets wrt to when we use EMTopo jets. This can be done using the AddFriend method for the trees.

Here you have a simple root macro you can use to test this feature on the files you just produced

```
#include "TFile.h"
#include "TTree.h"
#include <iostream>

using namespace std;

int compareMET()
{
    TFile *f = TFile::Open("output-combined.root","READ");
    TTree * Topo_truth = (TTree*)f->Get("EMTopo_truth");
    TTree * PFlow_truth = (TTree*)f->Get("EMPflow_truth");
    TTree * Topo_nominal = (TTree*)f->Get("EMTopo_nominal");
    TTree * PFlow_nominal = (TTree*)f->Get("EMPflow_nominal");

    Topo_truth->AddFriend(Topo_nominal);
    Topo_truth->AddFriend(PFlow_nominal);
    Topo_truth->AddFriend(PFlow_truth);

    Topo_truth->BuildIndex("eventNumber");
    PFlow_truth->BuildIndex("eventNumber");
    Topo_nominal->BuildIndex("eventNumber");
    PFlow_nominal->BuildIndex("eventNumber");

    Topo_truth->Draw("EMTopo_nominal.met_met:EMPflow_nominal.met_met","EMTopo_nominal.jet_pt@.size(>1 && EMPflow_nominal.jet_pt@.size(>1");

    return 1;
}
```

The first lines are clearly just to open the file and read the trees. The "AddFriend" option is the one telling ROOT that we plan to run in a synchronized way on these trees: as you see in the example, you can do that for more than just two. The [BuildIndex](#) is the method telling ROOT how to do the synchronization: in this case we are telling ROOT that we expect that entries having the same eventNumber belong to the same event.

This part in real life is a bit tricky, as analysers must know well what they're doing here: the quantity we use to build the index for the synchronization is expected to be an unique identifier for a given event in a given MC sample. Unfortunately, sometimes bug happens and this is no longer true, as discussed for example in this [talk](#). For the following of the tutorial, we'll forget about these problems (which hopefully will not happen often and will be solved quickly), but the lesson is: follow top-reco meeting and announcements on mailing list/inside the top digest, to make sure you're aware of possible bugs/problems.

Finally, as you see in the "Draw" method you can just access to the variables in the different trees by using their name as a prefix: "EMTopo\_nominal.met\_met" is the variable met\_met in the EMTopo\_nominal tree. The Draw function will automatically loop synchronizing the two trees. the second argument of the Draw function is an example of a selection that can be applied: in this case we're asking that events must have at least two jets in the two trees we're inspecting. This is also another useful trick: with the "@" command one can access to the size of vectors stored in the trees.

Now, just copy this code in a file called compareMET.C and run it

```
root -l compareMET.C
```

So now we have very easily a 2D plot showing for each event the MET when using PFlow jets vs the one obtained when using Topo jets!

You can find some more documentation on running on trees in a synchronized way [here](#).

## 12.4 Doing this on the GRID

It's easy to use this feature when running on the grid. You can just modify the submission script described above. Copy that in a new file grid/SubmitToGrid\_multiple\_example.py

```
#!/usr/bin/env python
import TopExamples.grid
import Data_rel21
import MC16_TOPQ1

config = TopExamples.grid.Config()
config.code = 'top-xaad'
config.settingsFile = 'example-config.txt example-config-topo.txt' # path to AT configuration file
config.combine_prefixes = "PFlow,Topo"
config.combine_outputFile = "out.root"
config.gridUsername = 'username' # username, or for group production e.g. phys-top or phys-higgs
config.suffix = 'test-multipleConfig' # mandatory suffix of the output dataset name (remember, that each job must have unique name)
config.excludedSites = '' # comma-separated list of sites to exclude from running our jobs
config.noSubmit = False # if True, will only simulate submission to grid, useful for testing
config.destSE = '' #This is the default (anywhere), or try e.g. 'UKI-SOUTHGRID-BHAM-HEP_LOCALGROUPDISK'
config.memory = '2000' # amount of RAM requested for the job (NOTE: scout jobs are used to determine the actual RAM consumption)
config.maxNFilesPerJob = '10' # each job in a pandatask will process no more than 10 input files
config.reuseTarBall = False # compress the files for grid submission once and reuse the tarball for all submissions (NOTE: will ignore any changes to

names = ["TOPQ1_ttbar_PowPy8_mc16a"] # define a list of sample groups (see MC16_TOPQ1.py)
samples = TopExamples.grid.Samples(names) # load the samples
TopExamples.ami.check_sample_status(samples) # this checks whether the samples exist on AMI, whether we use latest p-tag,...
TopExamples.grid.submit(config, samples) # submit the samples with the configuration above
```

Of course, do not forget to set the username to your own username.

As you see, there are three changes:

- config.settingsFile now is set to a comma-separated list of the config files we want to use
- config.combine\_prefixes is set to a comma-separated list of the prefixes we want the corresponding trees to have (the order MUST be the same as for the config files)
- config.combine\_outputFile is the name of the output combined file

If you want to actually test this feature on the grid, you may want to comment out the "NEvents 500" lines in the config files, so you don't limit yourself to just run on 500 events per subjob (this happens more often than you may think...). You may also want to do a test on a subset of a dataset. This is possible with a change like this one:

```
#config.maxNFilesPerJob = '10' # each job in a pandatask will process no more than 10 input files
config.otherOptions = '--nFiles=10 --nFilesPerJob=2'
```

This will run on just 10 files of the dataset, running on 2 files per job (so, we will have 5 subjobs). This can be useful to perform a quick test, instead of directly running on the whole stat, which is usually a bad idea if you're not sure the whole setup is working correctly.

## 13 What if I need to change the AnalysisTop code?


We will discuss this [in the advanced tutorial](#). There is also [this guide](#) on how to do a sparse checkout to be able to modify locally the AnalysisTop code (and in case to then merge it into the central repository).

### Major updates:

-- [MarcoVanadia](#) 07/04/2020

Responsible: [MarcoVanadia](#)

Last reviewed by: **Never reviewed**

I	Attachment	History	Action	Size	Date	Who	Comment
	<a href="#">AT.png</a>	r1	<a href="#">manage</a>	60.4 K	2020-04-16 - 19:52	<a href="#">MarcoVanadia</a>	The <a href="#">AnalysisTop</a> code in gitlab

Topic revision: r28 - 2020-05-12 - [OliverMajersky](#)