Steven Schramm
(Université de Genève)

Jet reconstruction: hands-on sessions

PREFIT School
March 2-13, 2020

Overview

This tutorial consists of two exercises. You may not finish all of them during the hands-on sessions. In that case, I would encourage you to finish it later at home! These two exercises cover very important jet reconstruction scenarios both from an experimental and more general perspective:

1. Jets reconstruction from an experimental perspective: pileup, response, resolution, and similar

2. Jet reconstruction from a more general perspective: jet definitions, grooming, and substructure

The steps are independent, and can be done in any order. Only the second step requires the FastJet software package. Throughout the exercises, you should be able to run the program after every step. The code is configured such that you can specify which step you want to run up to, both for the core code and the associated plotting scripts. Examples will be given in the following detailed descriptions.

Preparation

In this session, we will use ROOT to study jet properties (both exercises) and the FastJet software package to reconstruct new types of jets (second exercise). The input data will also be in ROOT format. You will thus need to have both ROOT and FastJet installed in order to complete this hands-on exercise.

1. ROOT can be downloaded from this page. The binary distribution is typically the most convenient. Please do this before the session, as it can take a while to install ROOT.

   – After you download ROOT and unpack the folder it comes in, you will find instructions for how to install it in the file named `INSTALL` within the `README` subdirectory.

2. FastJet is split between "core" and "contrib(uted)" packages. We will need both. Please do this in advance, as while it typically takes only a few minutes to install, there can be complications.

   – The core package can be downloaded and installed as described here [direct download link here]

   – The contrib package can be downloaded and installed as described here [direct download link here]

     ∗ Note: there is a new version as of March 7, you can use either the old version (1.043) or the new version (1.044). Both work, just make sure to update the commands below as appropriate.

   – Step-by-step instructions for installing fastjet and fastjet-contrib:

     1. Download fastjet and fastjet-contrib to a directory of your choice. Let's assume you decided to store it in a folder named "FastJet" in your home directory: `~/FastJet`
     2. Open a terminal and change to that directory: `cd ~/FastJet`
     3. Extract the fastjet files from the archive: `tar -zxvf fastjet-3.3.3.tar.gz`
     4. Extract the fastjet-contrib files from the archive: `tar -zxvf fjcontrib-1.044.tar.gz`
     5. Change to the extracted fastjet directory: `cd fastjet-3.3.3/`
     6. Configure fastjet to install in a new folder: `./configure --prefix=$PWD/../fastjet-install`
     7. Compile fastjet: `make`
     8. Check the fastjet compilation: `make check`
     9. Install the fastjet compilation: `make install`
     10. Change to the fastjet-contrib directory: `cd ../fjcontrib-1.044/`
     11. Configure fastjet-config to compile using the same config as fastjet:
         `./configure --fastjet-config=$PWD/../fastjet-3.3.3/fastjet-config`
     12. Compile fastjet-contrib: `make`
     13. Check the fastjet-contrib compilation: `make check`
     14. Install the fastjet-contrib compilation: `make install`
     15. You're done! Now fastjet and fastjet-contrib are installed in `~/FastJet/fastjet-install`

The dataset

The dataset used for both exercises is the same. This dataset is a preliminary release of a new ATLAS open dataset intended **exclusively for educational uses**. Detailes are below:

- ATLAS Pythia8 dijet MC samples, leading truth $R = 0.6$ jet p_{T} in the range of $[15\,\mathrm{GeV}, 2000\,\mathrm{GeV}]$
- The sample has been biased to have reasonable statistics of events out to high p_{T}. To recover the standard model dijet spectrum, an EventWeight must be applied as described in the exercises.
- For now, you can find 100k events here, password = JetReco
  - A full release with 1M events will follow soon on the CERN OpenData portal

Starting code and example solutions

In order to help you get started, code is provided which already has created all of the necessary histograms and set the necessary branches to read data out of the input ROOT file. Your task is to complete the programs such that they apply selections, build jets, or otherwise make choices before filling the specified histograms. This is denoted in the starting code files with comments that start with "TODO". If you want to learn more about how to read from ROOT TTrees, or how to create histograms and plots, I would encourage you to look instead at my ROOT tutorial available here (password = HASCO).

Plotting code is also provided alongside the starting code. The idea is that the primary code should run over the above-described dataset, producing a set of histograms. The plotting script then makes plots out of these histograms, including legends and overlaying results as appropriate to best display the results. If you are following the exercises, you should never need to modify the plotting code, you should just run it to see the results of your work.

Solutions are also provided, both in case you get really stuck or you want to work on this after the hands-on sessions. However, I would strongly recommend that you try to follow the exercise instructions first and only look at the solutions later in case of problems.

- Exercise 1: jet reconstruction and experimental considerations
  - Code folder: here, password = JetReco
  - Starting code: `jetRecoExp.cpp` (in the above link)
  - Plotting code: `jetRecoExp_plots.cpp` (in the above link)
  - Solution code: in the `solutions` folder of the above link
    - `jetRecoExp_solution.cpp`: the complete source code
    - `jetRecoExp_solution.root`: the root file produced by the complete source code
    - `jetRecoExp_solution.pdf`: the pdf file made by the plotting code run on the solution root file
- Exercise 2: jet reconstruction, grooming, and substructure
  - Code folder: here, password = JetReco
  - Starting code: `jetRecoGroom.cpp` (in the above link)
  - Plotting code: `jetRecoGroom_plots.cpp` (in the above link)
  - Solution code: in the `solutions` folder of the above link
    - `jetRecoGroom_solution.cpp`: the complete source code
    - `jetRecoGroom_clusters.root`: the root file produced by the complete code run on cluster jets
    - `jetRecoGroom_clusters.pdf`: the pdf file made by the plotting code run on the above file
    - `jetRecoGroom_truth.root`: the root file produced by the complete code run on truth jets
    - `jetRecoGroom_truth.pdf`: the pdf file made by the plotting code run on the above file

Exercise 1: jet reconstruction and experimental considerations

In this exercise, we will be studying experiment-related aspects of jet reconstruction. In particular, we will be comparing "reconstructed" jets (those which are built from objects observed in the detector) to "truth" jets (those built from stable particles independent of the detector). This comparison will help us to understand some of the experimental challenges faced by modern particle physics experiments when working with jets.

The code is already setup for you to handle reading from the input file and writing to the output file. All you need to do is work with the jet variables and fill the histograms as described below.

- **Step 0: getting started**
  - Install ROOT, if you have not done so already
  - Download the dataset, as described in the dataset section
  - Download the starting file `jetRecoExp.cpp` as described in the code section
  - Compile the starting file to make sure everything is setup correctly
    - Note that ‘ is a back-quote, which is different from the more commonly used apostrophe ’
    - If you can't find this key on your keyboard, you can also copy the command from the top of the `jetRecoExp.cpp` file (line 7 of the code)

  ```
  g++ jetRecoExp.cpp -o jetRecoExp ‘root-config --cflags --libs‘
  ```

  - Download the plotting script `jetRecoExp_plots.cpp` as described in the code section
  - Compile the plotting script to make sure everything is setup correctly

  ```
  g++ jetRecoExp_plots.cpp -o jre_plots ‘root-config --cflags --libs‘
  ```

  - That's it - you're ready to get started
- **Step 1: event-level information**
  - This is a simple first step just to get you familiar with the process
  - On line 219 of `jetRecoExp.cpp`, you will find the first "TODO" statement. Here, we have to fill three histograms using two key measures of pileup. One is the average number of interactions per beam crossing, typically referred to as μ. The other is NPV, the Number of observed (reconstructed) Primary Vertices in the event, as evaluated using the tracking detector.
  - The variables for these two quantities are defined on lines 69 and 70 of the file. They are `mu_average` (a floating point number) and `npv` (an unsigned integer). Note that you could have found these lines by searching for "Step 1" in the file, as a comment is directly before their definitions to help you navigate the file.
  - Now that we have the variables, the question is what histograms we need to fill. For the full definition, you can see lines 147-149, which also are preceeded by a comment stating that this is for "Step 1". However, for convenience, the names of all relevant histograms are also written directly beside the area that they need to be used. As such, going back to line 219 and looking just above it lists the names of the three histograms: `hist_mu`, `hist_npv`, and `hist_mu_npv`.
  - We now have everything we need. We can fill the histograms directly by adding the following code in the area of line 219:

  ```
  if (!stepNum || stepNum >= 1)          // already exists
  {                                      // already exists
      hist_mu.Fill(mu_average);          // new code
      hist_npv.Fill(NPV);                // new code
      hist_mu_npv.Fill(mu_average,NPV);  // new code
  }                                      // already exists
  ```

  - With this done, we have completed the first step. First, re-compile the program as described in step 0, and then run the code saying that we want step 1. The program has the following arguments:
    - The first argument is the output ROOT histogram file (we chose to call it `jetRecoExp.root`)
    - The second argument is the step we want to run up to (1 in this case)

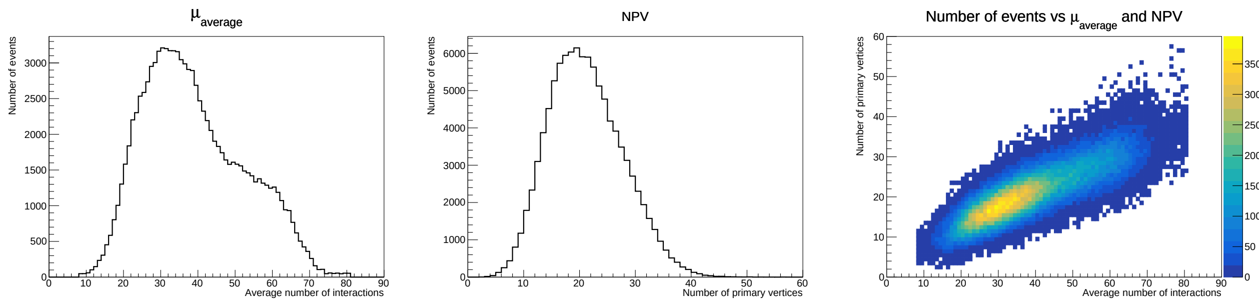
    * The third argument is the name of the tree from the input file, which is `JetRecoTree`
    * The fourth argument is the name of the file that we want to read from and that you downloaded earlier, which by default is named `JetRecoDataset.root`
  - Putting this all together, we get the following command:

```
./jetRecoExp jetRecoExp.root 1 JetRecoTree JetRecoDataset.root
```

  - After running this command, we have a new file named `jetRecoExp.root` which contains a couple of histograms. You can open this file with ROOT to study the contents if you want, but you can also run the provided plotting script to quickly see the results in a useful format. To do that, we have to run the plotting script which has the following arguments:
    * The first argument is the output pdf plot file (here we have chosen to call it `jetRecoExp.pdf`)
    * The second argument is the step we want to run up to (1 in this case)
    * The third argument is the input ROOT histogram file, which is `jetRecoExp.root` in this case
  - Putting this all together, we get the following command:

```
./jre_plots jetRecoExp.pdf 1 jetRecoExp.root
```

  - You should now have a pdf file with the following three plots, each on a separate page:



  - This shows the clear correlation between the two pileup measures, but that they are not identical. There is a spread of values, as one is an average quantity over many events while the other is what was observed in a given event. The values of `mu_average` and `NPV` are typically used to represent *out-of-time* and *in-time* pileup respectively.
    * In-time pileup (`NPV`): the number of "simultaneous" proton-proton collisions in a beam crossing. This is the primary metric for tracking detectors and similar where individual bunch crossings can be resolved.
    * Out-of-time pileup (`mu_average`): the number of collisions expected on average over many beam crossings. This is the primary metric for calorimeters and similar where the amount of time it takes to read out the signal from a given bunch crossing may be longer than the gap between bunch crossings, and thus signals from different subsequent beam crossings overlap.
    * Ultimately, jets are sensitive to both of these pileup metrics and in different ways
  - That's the end of the first step. The following steps will provide less details, but the same principle applies. You should:
    * Look for the "Step N" comments in the code, where N is the current step you are working on
    * The histograms you have to fill will be also listed by "Step N" and "TODO" comments
    * The variables you need to use to do this have already been read in, and you can find them between lines 68 and 138 next to the respective "Step N" comments

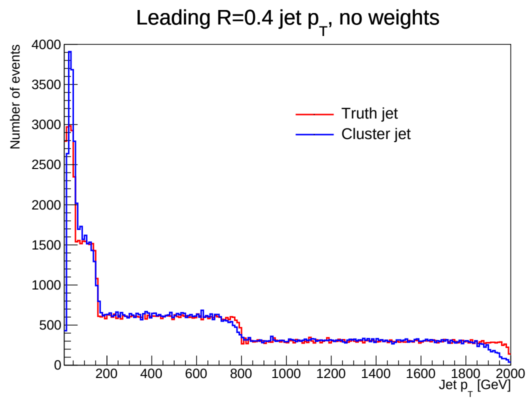
- **Step 2: R=0.4 cluster and truth jets and the event weight**
  - Next, we want to look at the p_{T} distribution of the leading (highest-p_{T}) jet. We want to do this for both calorimeter (reconstructed) jets and truth (detector-independent) jets.
  - We want to do this both with and without a so-called `EventWeight`
    * Without such a weight, you see the raw statistical power of the sample and the number of events as a function of p_{T}. This will not look like a physical distribution, but rather is related to how the sample was produced such that we still have events at high p_{T}.

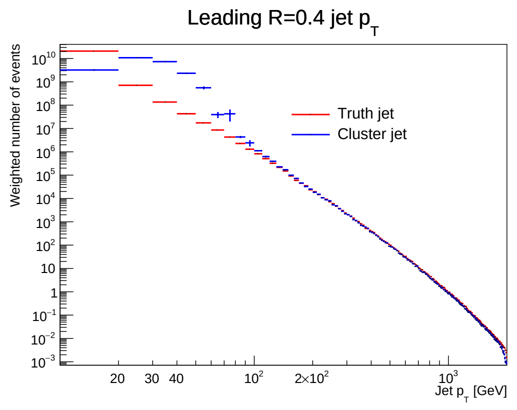
    * With such a weight, you should recover the expected physical spectrum of QCD multijet production, which is a quickly falling distribution.

- To do this, we can again use the variables that have already been setup for our use:
    * `RecoJet_pt`: a `vector<float>` of the p_T of all of the jets in the event, sorted such that the first jet (index 0) is the highest p_T jet. This is for reconstructed calorimeter jets.
    * `TruthJet_pt`: the same, but for truth jets.
    * `EventWeight`: a `float` representing the factor that re-creates the physical distribution
- We can then use these variables, making sure to only consider events where at least one jet exists:

```
if (RecoJet_pt->size())
{
  hist_reco_pt_nw.Fill(RecoJet_pt->at(0)); // No weights (nw)
  hist_reco_pt.Fill(RecoJet_pt->at(0),EventWeight); // Weighted
}
if (TruthJet_pt->size())
{
  hist_truth_pt_nw.Fill(TruthJet_pt->at(0)); // No weights (nw)
  hist_truth_pt.Fill(TruthJet_pt->at(0),EventWeight); // Weighted
}
```

- This completes the second step. If you re-compile the code, run it specifying step 2, and also run the plotting script specifying step 2, you should now get an additional two plots:





- These plots make it clear that the calorimeter (cluster) and truth jets agree reasonably well at high p_T, but a significant disagreement starts to appear for $p_\mathrm{T} < 100\,\mathrm{GeV}$. Note that the second plot is on a logarithmic y axis, so the differences are very large!

- **Step 3: Pileup dependence**
    - In the last step, we have seen our first evidence of pileup. The truth jets are only from the vertex of interest, while calorimeter cluster jets can confuse signals from both out-of-time sources or additional in-time collisions. In this step, we will study this pileup dependence in more detail.
    - The code is now becoming more complex, as we want to make more advanced comparisons. First, let's count the number of jets there are above 20 GeV:

```
  // Count the number of cluster jets
  unsigned numJetReco = 0;
  for (size_t iJet = 0; iJet < RecoJet_pt->size(); ++iJet)
  {
    if (RecoJet_pt->at(iJet) > 20.e3)
      numJetReco++;
  }
```

    - We now have the jet multiplicity for a 20 GeV p_T threshold in a given event. Let's study how this value evolves as a function of `mu_average`, one of our pileup-quantifying variables. Let's only consider events that contain at least a single jet, as those are more interesting to us. We can then divide the events into three bins of `mu_average`. I would recommend using bins of [0,30], [35,45], and $[50,\infty]$ so that each bin has a similar statistical power (see the plots from Step 1 for an idea of how these values were chosen).

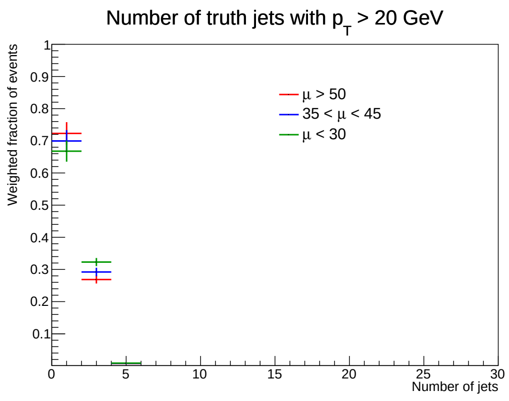
```
// We only want to consider events that have at least one jet
if (numJetReco != 0)
{
  if      (mu_average < 30)
    hist_reco_njets_lowmu.Fill(numJetReco,EventWeight);
  else if (mu_average > 35 && mu_average < 45)
    hist_reco_njets_midmu.Fill(numJetReco,EventWeight);
  else if (mu_average > 50)
    hist_reco_njets_highmu.Fill(numJetReco,EventWeight);
}
```

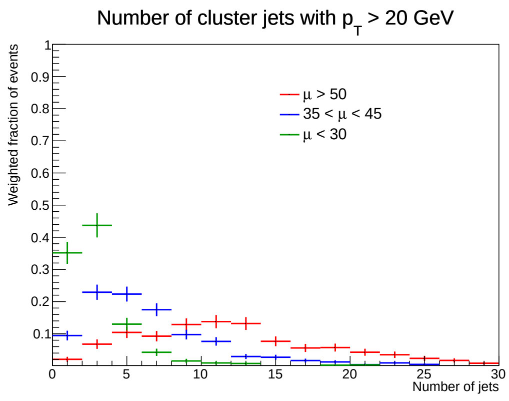
– While we are plotting these slices, let's also plot the full dependence on both `mu_average` and `NPV`. We can do this by adding one additional line to the above, within the check to ensure that there is at least one jet:

```
hist_reco_njets_mu_npv.Fill(mu_average,NPV,numJetReco,EventWeight);
```

– At this point, we have finished what we need to do for the reconstructed (calorimeter cluster) jets. Now, repeat the same thing for truth jets, which can generally be done by replacing every instance of "reco" or "Reco" in the above with "truth" or "Truth" respectively.

– With that done, with have finished the third step. If you re-compile the code, run it specifying step 3, and also run the plotting script specifying step 3, you should now get an additional four plots. The first two are as follows:

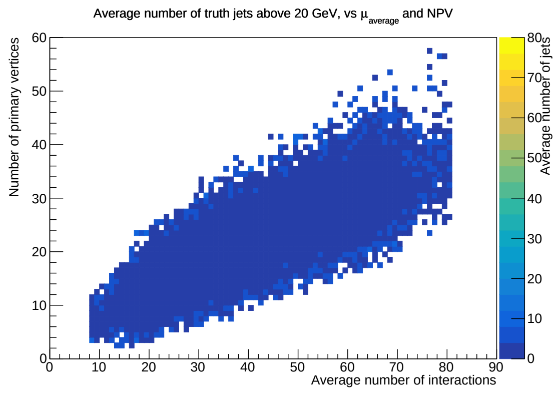


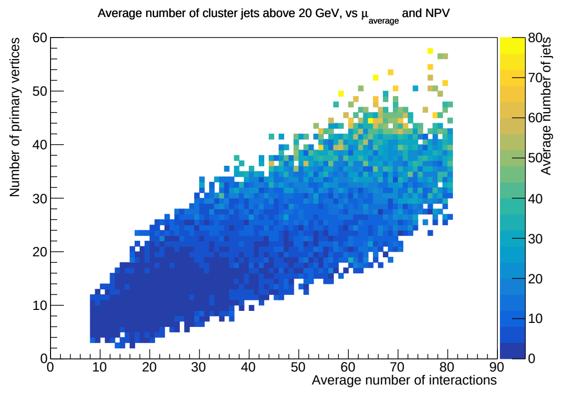


– This makes it clear that the number of truth jets is quite stable as a function of pileup (left plot), while the calorimeter jets are varying dramatically (right plot). This is actually showing us two different effects:

  * There are more jets in the detector due to both in-time and out-of-time pileup
  * Jets from the collision of interest can also have their energy scale increased through the addition of stochastic overlaps of energy from different collisions. In this case, there will be more jets as they are more likely to have a p_T above our chosen threshold of 20 GeV.

– We can see this as well from the second set of new plots produced, where `mu_average` and `NPV` are separated, showing the increase of calorimeter cluster jet multiplicity vs both variables:





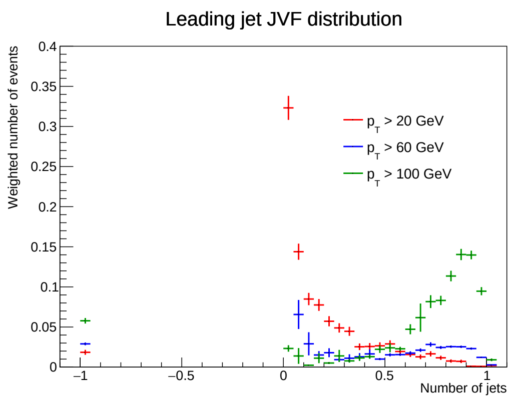
- **Step 4: Tracks and R=0.4 track jets**
  - I mentioned earlier that the calorimeter is sensitive to out-of-time pileup, while the tracker is not as the read-out time is shorter than the time between subsequent beam crossings. Trackers are even more powerful than that, as they can also identify the origin of a particle as coming from one specific collision vertex instead of another simultaneous collision vertex. Trackers are thus enormous useful in suppressing pileup contributions.
  - One way to use tracking information to help the calorimeter is to match tracks to calorimeter jets and then calculate the Jet Vertex Fraction (JVF). The JVF calculation is defined as follows:

$$\text{JVF} = \frac{\displaystyle\sum_{i\in\{\text{HS}\}} p_{\text{T}}^{i}}{\displaystyle\sum_{t\in\{\text{PV}\}} p_{\text{T}}^{t}} \qquad \begin{array}{l}\text{HS} = \text{all tracks matched to the jet from the hard scatter primary vertex}\\ \text{PV} = \text{all tracks matched to the jet from any primary vertex}\end{array}$$

  - This variable has already been calculated, and can be accessed with `RecoJet_jvf`, a `vector<float>`
  - Before using this variable to suppress pileup, let's take a look at it by filling histograms of this variable with calorimeter jet p_{T} cuts of 20 GeV, 60 GeV, and 100 GeV. An example for filling the 20 GeV histogram is below, you then have to modify this for the 60 GeV and 100 GeV histograms:

```
if (RecoJet_jvf->size() && RecoJet_pt->at(0) > 20.e3)
  hist_reco_jvf_pt20.Fill(RecoJet_jvf->at(0),EventWeight);
```

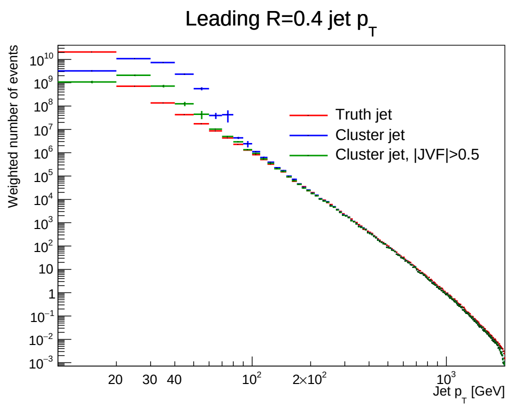
  - The resulting plot shows how the JVF distribution shifts from values of 0 (pileup-like) to 1 (hard-scatter-like) as the p_{T} increases. Note that there is also a population of jets with a JVF value of -1, which corresponds to calorimeter jets that have zero tracks pointing at them. This could be either from jets in the forward region beyond the tracker acceptance, purely from out-of-time pileup (where there are no tracks since the tracker is only for the beam crossing of interest), or from real physical processes that leave no charged energy contributions (π^0 dominated showers, neutrons, etc).



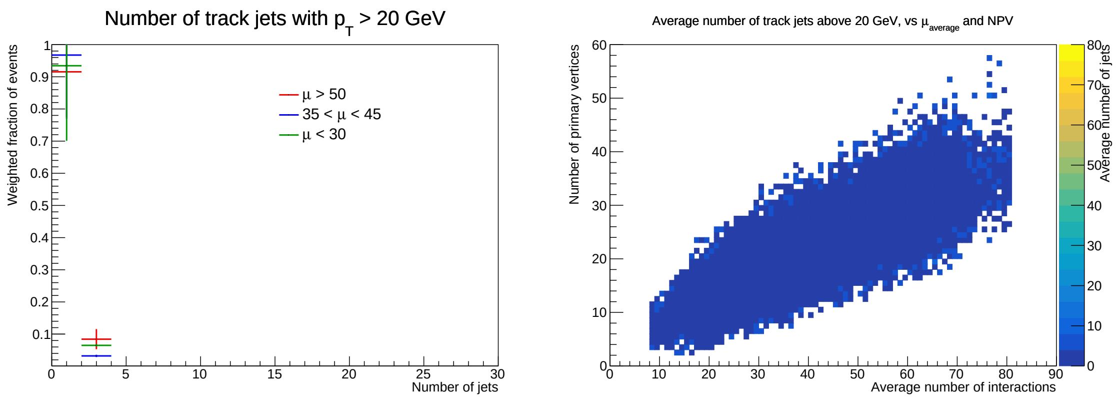
  - A reasonable first value to consider as a JVF cut is 0.5, which means that at least half of the track momentum pointing at a jet comes from the hard-scatter vertex. Note that it is usually better to apply the cut on the absolute value of JVF, or $|\text{JVF}| > 0.5$, in order to not remove jets with zero tracks, for the reasons mentioned earlier. To do this, we will apply a JVF cut as follows and fill a new leading jet p_{T} histogram:

```
if (RecoJet_jvf->size() && fabs(RecoJet_jvf->at(0)) > 0.5)
  hist_reco_pt_jvf.Fill(RecoJet_pt->at(0),EventWeight);
```

  - With this done, we will get the following plot. This shows that JVF is doing a good job of resolving the challenges that we have using calorimeter jets at low p_{T}. Again, note that the plot has a logarithmic y-axis, and thus this is really an enormous improvement even if this first simple cut isn't sufficient to completely fix the problem.



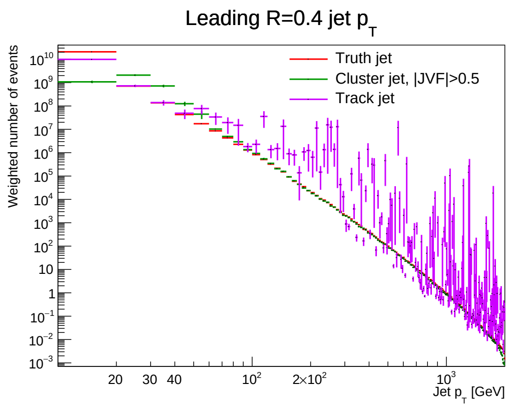
- We have seen how tracks can improve the pileup-robustness of our calorimeter jets. However, what do track jets themselves look like? After all, we can build jets out of any set of four-vectors, so we can build them using tracks only from the hard-scatter vertex that we are actually interested in.
- Let's do this, using the `TrackJet_pt` variable, which is also a `vector<float>`. This variable is the pT of a set of jets that were built using only tracks, and those tracks were only used if they came from the hard-scatter vertex. Repeat the exact same code as you did for Step 3, but now switch to using track jets for the p_{T} cuts and multiplicity. Once done, you will get plots like the following, which make it clear that track jets are very stable against pileup effects.



- The question then naturally arises: if track jets are so pileup-robust, why don't we use them instead of calorimeter jets? To start to answer that question, let's simply plot the track jet p_{T} distribution

```
if (TrackJet_pt->size())
  hist_track_pt.Fill(TrackJet_pt->at(0),EventWeight);
```

- Comparing the track jet p_{T} to the truth or calorimeter jet p_{T} shows much more fluctuations, which starts to hint at why we don't use track jets.

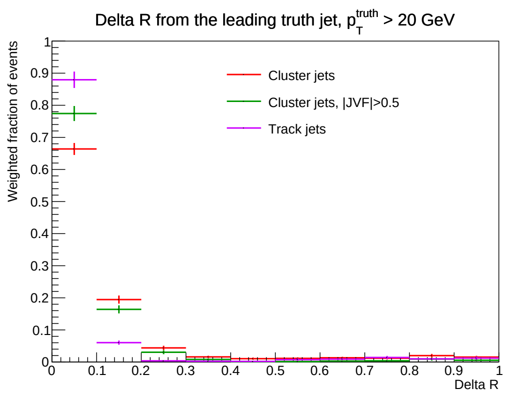


- We have now seen that we can use tracks together with calorimeter jets to help suppress pileup, and that using tracks alone is very pileup-robust. However, we have also seen some indications that using tracks by themselves is not necessarily a good choice. To understand why, we need to proceed to the last step.

- **Step 5: Jet response studies**
  - In order to better understand what is going on, we need to directly compare truth jets to reconstructed jets (either track jets or calorimeter jets). To do this, we can "match" truth and reconstructed jets, using ΔR (Delta R), the angular distance between the truth jet axis and reconstructed jet axis. We will always use the truth jet as our reference point since we will match to multiple types of reconstructed jets and we want to compare them fairly. As such, we can define the Delta R between the leading truth and reconstructed calorimeter jet as follows (then you can do the same for the leading track jet, but without the extra jvf-cut histogram):

```
if (TruthJet_pt->size() && TruthJet_pt->at(0) > 20.e3)
{
  TLorentzVector truthJet;
  truthJet.SetPtEtaPhiM(TruthJet_pt->at(0),TruthJet_eta->at(0),
                        TruthJet_phi->at(0),TruthJet_m->at(0));

  if (RecoJet_pt->size())
  {
    TLorentzVector recoJet;
    recoJet.SetPtEtaPhiM(RecoJet_pt->at(0),RecoJet_eta->at(0),
                         RecoJet_phi->at(0),RecoJet_m->at(0));

    hist_DRtruth_reco.Fill(truthJet.DeltaR(recoJet),EventWeight);
    if (fabs(RecoJet_jvf->at(0)) > 0.5)
      hist_DRtruth_reco_jvf.Fill(truthJet.DeltaR(recoJet),
                                 EventWeight);
  }
}
```

  - With this done for both calorimeter and track jets, the following plot is obtained:

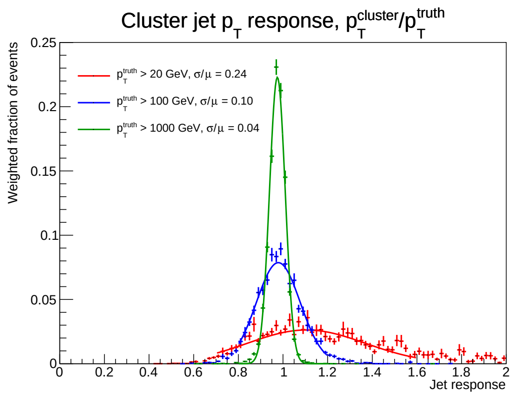


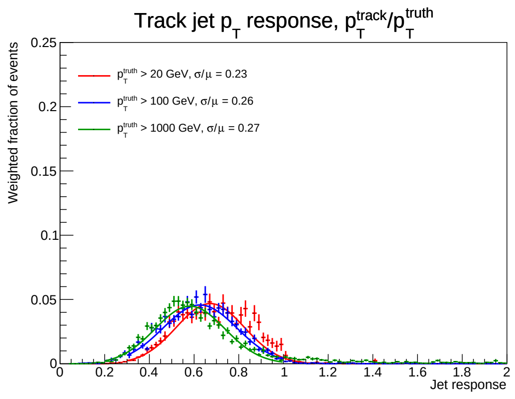
  - Here, it is clear that track jets slightly better match to the truth jet. This isn't too surprising given that we are only applying a 20 GeV cut here, where we saw that calorimeter jets have limitations due to pileup. Additional random energy overlapping with the jet can easily change the jet axis, leading to a larger value of Delta R.
  - Additionally, we are only plotting the range very close to the truth jet (Delta R < 1). If you edit the code to plot larger values of Delta R, you will see that many jets (both calorimeter and track) have Delta R values of 2 or larger. This is expected when the leading and subleading jets switch, and we could avoid it by looping over all of the reconstructed jets to find the closest in Delta R, but we are not doing so here for simplicity.
  - Despite the above disclaimers, we can see that there is a pretty good match in many cases. There are many events where the Delta R value is below 0.3. Let us use this as a matching cut, meaning that we say that a truth jet matches a reco jet if DeltaR(truth,reco) < 0.3.
  - Now, use matched truth-to-reco jets to study the response, which is the ratio the reconstructed to truth jet p_T for a matched pair of jets, $\mathcal{R} = p_T^{reco}/p_T^{truth}$

```
if (truthJet.DeltaR(recoJet) < 0.3)
{
  hist_response_reco_pt20.Fill(recoJet.Pt()/truthJet.Pt(),
                                            EventWeight);
}
```

– The above is for a 20 GeV truth jet p_T threshold, as we previously required a 20 GeV truth jet. Now, do the same thing also for truth jet p_T thresholds of 100 GeV and 1000 GeV to fill the other two histograms. Then, do this same thing for track jets instead of calorimeter jets.

– Congratulations, you have finished step 5 and thus the exercise! The final results are as follows:





– Here, you can see the response for calorimeter jets (left) and truth jets (right). You can notice a few important points:

  ∗ These reconstructed jets we are using are uncalibrated, so they have different central values. This central value difference could be fixed so it is not inherently a problem. Note that track jets peak around a value of 2/3, which makes sense as jets are dominated by pion production and 2/3 of pions are charged (π^+ and π^- vs π^0).

  ∗ Also note that the calorimeter jets typically have a response larger than 1 at low p_T. This is because they have an underlying amount of energy added to them, mostly from pileup contributions. This disappears at higher energy where pileup contributions become small compared to the scale of the hard-scatter jet under study.

  ∗ Finally, and most important, note the difference in the width of the distributions. This is typically called the *resolution*. Normalizing the resolution by the different scales allows for removing effects related to being at different energies, and thus rough resolution values of σ/μ are shown for each line as a part of the legend.

  ∗ For calorimeter jets, the resolution dramatically improves as the p_T is increased. This is because the low p_T resolution is dominated by noise (mostly from pileup), which reduces at high p_T. The calorimeter also does a better job of measuring larger amounts of energy, further improving the resolution at high p_T.

  ∗ In contrast, track jet resolution is roughly stable and eventually degrades as the p_T is increased. This is because the tracker p_T resolution requires observing curvature of the tracks, which degrades at high p_T and thus so does the track jet resolution. Additionally, track jets lack the neutral contributions which is roughly 1/3 of the typical jet energy, and thus there is an inherently large resolution coming from this missing energy contribution.

• That's it! You have finished the first exercise.

Exercise 2: jet reconstruction, grooming, and substructure

In this exercise, we will be studying large-R jets and the impact of different grooming choices on key variables. In particular, we will be looking at the jet p_{T} (always relevant), jet mass (relevant for most jet tagging), $D_2^{\beta=1}$ (relevant for W/Z boson tagging), and τ_{32}^{WTA} (relevant for top-quark tagging). As we only have QCD multijet samples, we will not be designing actual taggers, but rather studying the impact on the background (QCD) when we make different grooming choices for reconstructed large-R jets. If you are interested, it is also easy to do the same thing for truth jets at the end, so you can compare truth and reconstructed jets for key variables.

In this case, you will need to use fastjet, as you will be building jets from calorimeter clusters. When you get to that step, it is normal for the code to slow down quite a bit. Fastjet itself is very fast, but building many jets for 100k events takes a while. On my laptop, it took a minute or two to run over the full sample up to step 4. Step 5 adds the $D_2^{\beta=1}$ and τ_{32}^{WTA} calculations, which are very CPU expensive, and so it slowed down the process to roughly five minutes to run over the full sample. Nonetheless, you do need to run over all of the events in the file. The events are not distributed randomly, so you can't run over 1/10th of the file and expect to see events everywhere just with lower statistics. You need to run over the full file to make sure that you will cover the relevant kinematic regimes.

In this exercise, we will be using $R = 1.0$ jets, which are the ATLAS default for large-R jets. However, you are free to try any other size you want for the latter steps since you will be building your own jets. You can even compare different sizes of jets if you want, such as also looking at the CMS default of $R = 0.8$.

As before, the code is already setup for you to handle reading from the input file and writing to the output file. All you need to do is work with the jet variables and fill the histograms as described below.

- **Step 0: getting started**
  - Install ROOT, if you have not done so already
  - Install fastjet and fastjet-contrib, if you have not done so already
  - Download the dataset, as described in the dataset section
  - Download the starting file `jetRecoGroom.cpp` as described in the code section
  - Compile the starting file to make sure everything is setup correctly
    - Note that ‘ is a back-quote, which is different from the more commonly used apostrophe ’
    - If you can't find this key on your keyboard, you can also copy the command from the top of the `jetRecoGroom.cpp` file (line 7 of the code)
    - Also note a \ is used to say I ran out of space, the below should be a single-line command
    - You may have to update the command to point to your fastjet-config location

    ```
    g++ jetRecoGroom.cpp -o jetRecoGroom \
    `~/FastJet/fastjet-install/bin/fastjet-config --cxxflags --libs \
    --plugins` `root-config --cflags --libs` \
    -lRecursiveTools -lEnergyCorrelator -lNsubjettiness
    ```
  - Download the plotting script `jetRecoGroom_plots.cpp` as described in the code section
  - Compile the plotting script to make sure everything is setup correctly

    ```
    g++ jetRecoGroom_plots.cpp -o jetRecoGroom_plots \
    `root-config --cflags --libs`
    ```
  - That's it - you're ready to get started
- **Step 1: Event-level information**
  - Please see step 1 of exercise 1. It is essentially the same, and is intended to check that your code is working correctly. In this case we are only making two plots (μ and N_{PV}), not also the two-dimensional plot from exercise 1. The relevant code for the entire part of step 1 is as follows:

    ```
    if (!stepNum || stepNum >= 1)
    {
      hist_mu.Fill(mu_average);
      hist_npv.Fill(NPV);
    }
    ```

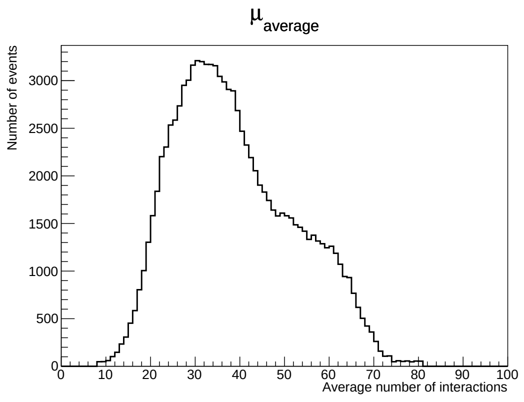
- The way to use the code is also the same as before. To run over the dataset (up to step 1):

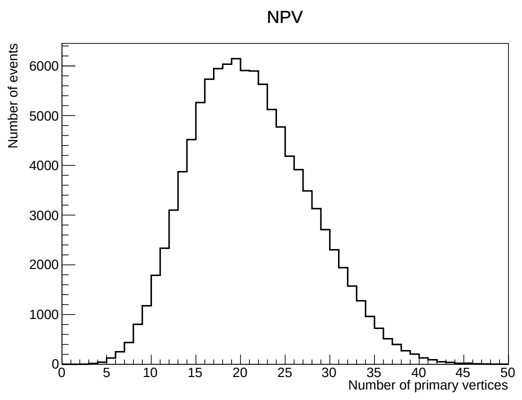
  ```
  ./jetRecoGroom jetRecoGroom.root 1 JetRecoTree JetRecoDataset.root
  ```

- To run the plotting script (for up to step 1):

  ```
  ./jetRecoGroom_plots jetRecoGroom.pdf 1 jetRecoGroom.root
  ```

- This should then result in the following two plots:



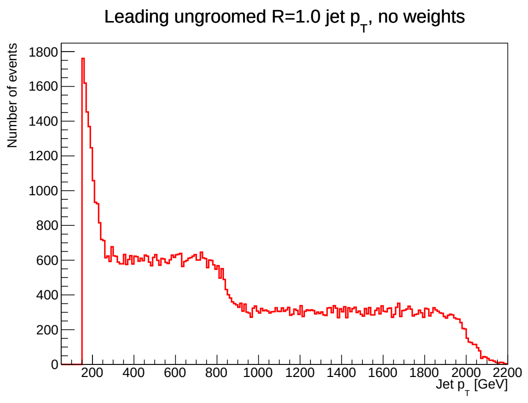


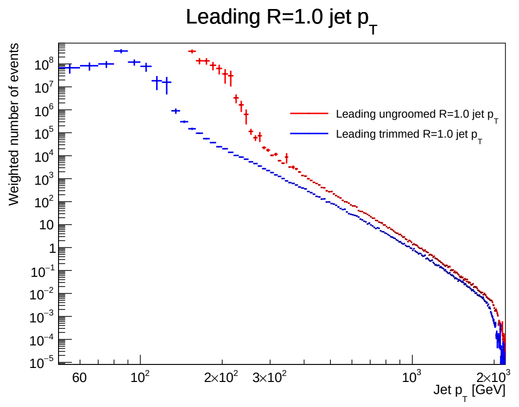
• **Step 2: Existing jets and the event weight**

- This is also similar to the second step of the first exercise, but with a few important differences. First, we want to use large-R jets instead of small-R jets, so the name of the variable to use is slightly different. Second, we want to look at both the original (ungroomed) large-R jets as well as the trimmed (ATLAS default) large-R jets. As such, the code becomes:

  ```
  if (jet_R10_ungroom_pt->size())
  {
    hist_ungroom_pt_nw.Fill(jet_R10_ungroom_pt->at(0));
    hist_ungroom_pt.Fill(jet_R10_ungroom_pt->at(0),EventWeight);
    hist_trimmed_pt.Fill(jet_R10_trimmed_pt->at(0),EventWeight);
  }
  ```

- Running the plotting script on these outputs produces the following pair of plots:



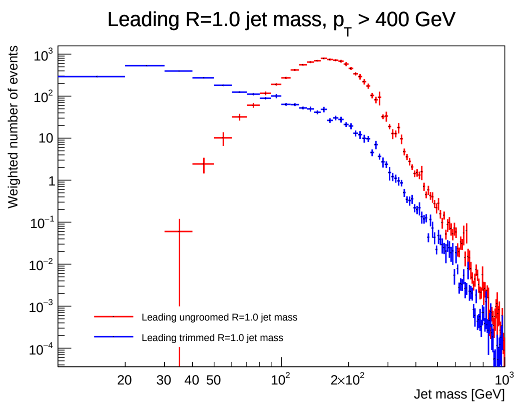


- The same unweighted plot as exercise 1 makes it clear that we still need to use the event weights to have a proper physical spectrum.
- Additionally, comparing ungroomed to trimmed jets is very interesting. There are two clear effects:
  * At low p_T, the smooth slope of the leading jet p_T spectrum suddenly changes. This happens at ∼300 GeV for the ungroomed jets, and ∼150 GeV for the trimmed jets. This change in behaviour at low p_T is from pileup. As we are using a very large jet radius of $R = 1.0$, we are much more susceptible to such effects than in the first exercise where we used $R = 0.4$, and thus the impact extends to much higher p_T.
  * Above this pileup regime, the ungroomed and trimmed collections still do not agree. This is because grooming also removes the *underlying event*, and other energy that is soft with respect to the total jet energy scale. This means that the groomed jet should always have lower energy (or at most equivalent energy) to the ungroomed jet.

- Now that we have looked at the leading jet p_{T} spectrum, let's also look at the mass of the same leading jet. Let's only look at jets which have at least 400 GeV of p_{T} in order to avoid the pileup regime. The relevant code to do this is:

```
if (jet_R10_ungroom_pt->at(0) > 400.e3)
  hist_ungroom_m.Fill(jet_R10_ungroom_m->at(0),EventWeight);
if (jet_R10_trimmed_pt->at(0) > 400.e3)
  hist_trimmed_m.Fill(jet_R10_trimmed_m->at(0),EventWeight);
```

- Running the code with this addition results in the following jet mass plot:



- Recall that this is a QCD multijet MC sample, so there is no true resonance that should result in a spike in the jet mass at ∼200 GeV. As such, we are seeing the impact of the underlying event promoting the jet mass up to very large values for the ungroomed jet. After trimming, the jet mass is dramatically suppressed, keeping in mind that the y-axis uses a logarithmic scale. Instead, the trimmed jet mass is most likely to occur at very low values of ∼30 GeV.
- This difference is very important for tagging, as the ungroomed distribution would hide the top quark mass while the trimmed distribution is much reduced in that regime. It is also important as it is now showing a more realistic representation of the QCD jet mass expectation.

- **Step 3: Building our own R=1.0 jets from topoclusters**
  - Now that we have retrieved jets from the file, it's time to build jets of our own. The next step makes use of fastjet, and requires reading a lot more data for each event. To get a feeling for how much this changes the data size, the average number of $R = 1.0$ jets saved in each event of the file is 2, while the average number of clusters saved in each event of the file is 590. As such, a much larger amount of data is being read out of the file in each event, which slows down the code dramatically. This is further slowed down by running jet reconstruction on all of those clusters. This is also why the dataset is roughly 2 GB in size: the jets take up very little space while the tracks, clusters, and truth particles take up a lot of space due to their large multiplicity.
  - When building jets of our own, it's useful to compare against existing jets to make sure that everything is configured correctly. The goal of step 3 is thus to reproduce the $R = 1.0$ ungroomed and trimmed jet collections which are already in the file and which we used in step 2. In order to do this, we need to do a few things. First, we need to add new fastjet headers to the very beginning of the file (search for "Step 3"):

```
#include "fastjet/ClusterSequence.hh"
#include "fastjet/tools/Filter.hh"
```

  - These header files provide the tools that we need to run jet reconstruction (build a cluster sequence) and then trim (or filter) the result.
  - Next, we need to configure the fastjet tools that we need to run jet reconstruction. This should be around line 215 of the file where you need to add the following:

```
fastjet::JetDefinition akt10(fastjet::antikt_algorithm,1.0);
fastjet::Transformer *trimmer = new fastjet::Filter(
        fastjet::JetDefinition(fastjet::kt_algorithm, 0.2),
        fastjet::SelectorPtFractionMin(0.05) );
```

– At this point, we have prepared the fastjet tools and configuration that we will need to build and then trim large-R jets.

– Finally, we need to convert our input clusters into a format that fastjet recognizes, namely the `fastjet::PseudoJet` class. This expects inputs in the form p_x, p_y, p_z, E while our clusters are instead stored in a different format of $p_{\mathrm{T}}, \eta, \phi, m$. We therefore have to convert our clusters to the fastjet format, which is convenient to do using the ROOT TLorentzVector (more info here).

  ∗ Note that in ATLAS, clusters have a mass of zero, so in case you run into this it's not a bug but rather because I kept the stored variables the same between all objects for consistency

– Here is a small piece of code that converts four vectors of floats, representing clusters, into a vector of `PseudoJet`. You will need to add this to the code within the "Step 3" area of the event loop.

```
// Convert the clusters into FastJet's four-vector (PseudoJet)
std::vector<fastjet::PseudoJet> clusters;
clusters.reserve(cluster_pt->size());
for (size_t iClus = 0; iClus < cluster_pt->size(); ++iClus)
{
  TLorentzVector cluster;
  cluster.SetPtEtaPhiM(cluster_pt->at(iClus),
                       cluster_eta->at(iClus),
                       cluster_phi->at(iClus),
                       cluster_m->at(iClus));
  clusters.push_back(fastjet::PseudoJet(cluster.Px(),
                                        cluster.Py(),
                                        cluster.Pz(),
                                        cluster.E()));
}
```

– We are now ready to run jet reconstruction. Thanks to fastjet, this is actually remarkably easy:

```
// Use fastjet to build new jets
fastjet::ClusterSequence cs_a10_clusters(clusters,akt10);
std::vector<fastjet::PseudoJet> jets_a10_clusters =
       fastjet::sorted_by_pt(cs_a10_clusters.inclusive_jets());
```

– That's it. Those two lines have taken all of the cluster inputs, run the jet algorithm on them, and given us back a vector of `PseudoJet` representing the $R = 1.0$ ungroomed jets! Furthermore, it has given us back the jets in a sorted order, with the first jet having the highest p_{T}.

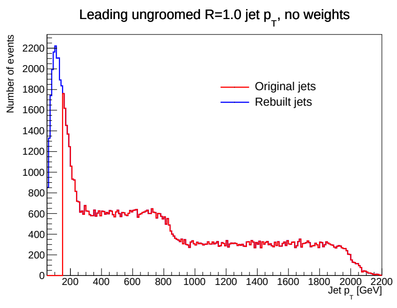
– Let's compare the leading (highest p_{T}) `PseudoJet` with what we had back in step 2. To do that, fill the histograms with your new $R = 1.0$ ungroomed jets:

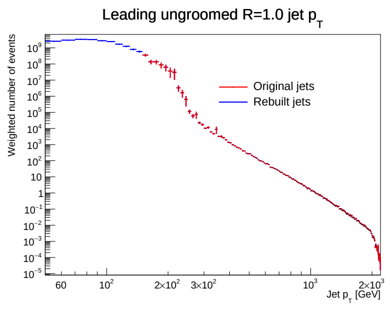
```
// Use these jets and compare to the original jets
if (jets_a10_clusters.size())
{
  const fastjet::PseudoJet& ungroomed = jets_a10_clusters.at(0);
  hist_myungroom_pt_nw.Fill(ungroomed.pt());
  hist_myungroom_pt.Fill(ungroomed.pt(),EventWeight);
}
```

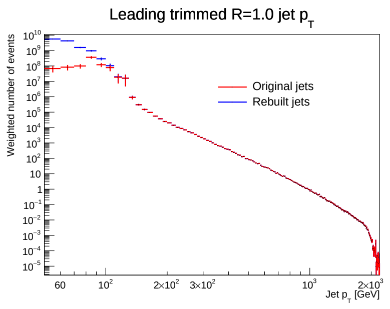
– At the same time, let's quickly run trimming on just the leading $R = 1.0$ jet, and then save that jet to a separate pair of histograms:

```
  // Trim the jet
  fastjet::PseudoJet trimmed = (*trimmer)(ungroomed);
  hist_mytrimmed_pt_nw.Fill(trimmed.pt());
  hist_mytrimmed_pt.Fill(trimmed.pt(),EventWeight);
```

- We now have built our own $R = 1.0$ jets from clusters using fastjet, and we have also trimmed the leading jet. Assuming everything has been configured correctly, the plotting script should now produce the following three plots:







- The first two plots make it very clear what has happened: the jets stored in the file are identical to the ones we just built, but the ones stored in the file are only stored if their p_{T} is above 150 GeV. This is indeed the case - the public dataset only contains $R = 1.0$ ungroomed jets that pass such a threshold. We have thus validated our implementation of fastjet for reconstructing ungroomed $R = 1.0$ jets.
- The third plot is a bit harder to interpret, but it's actually exactly the same effect. What was done when making the file is that the 150 GeV threshold was only applied on the ungroomed jet. The trimmed jet p_{T} was irrelevant to the decision of whether or not to store the jet: trimmed jets were stored whenever the ungroomed jet that they came from had a $p_{\mathrm{T}} > 500$ GeV. As the ungroomed jet has an equal or larger p_{T} than the trimmed jet, the trimmed jets should also agree by 150 GeV, which indeed they do. We have thus also validated our implementation of the trimming algorithm.
- We have now validated that we can build jets from clusters and then groom them. It's time to apply lots of different types of grooming strategies!

- **Step 4: Building other types of R=1.0 jets from topoclusters**
  - ATLAS has used trimming for many years, but there are now many other grooming options available.
  - CMS often made use of Pruning in the past, and now typically uses SoftDrop.
  - More recently, there have also been advanced SoftDrop variants released, such as Recursive or Bottom-Up SoftDrop.
  - We will consider these four alternative grooming methodologies, which require the following additional header files:

```
#include "fastjet/tools/Pruner.hh"
#include "fastjet/contrib/SoftDrop.hh"
#include "fastjet/contrib/RecursiveSoftDrop.hh"
#include "fastjet/contrib/BottomUpSoftDrop.hh"
```

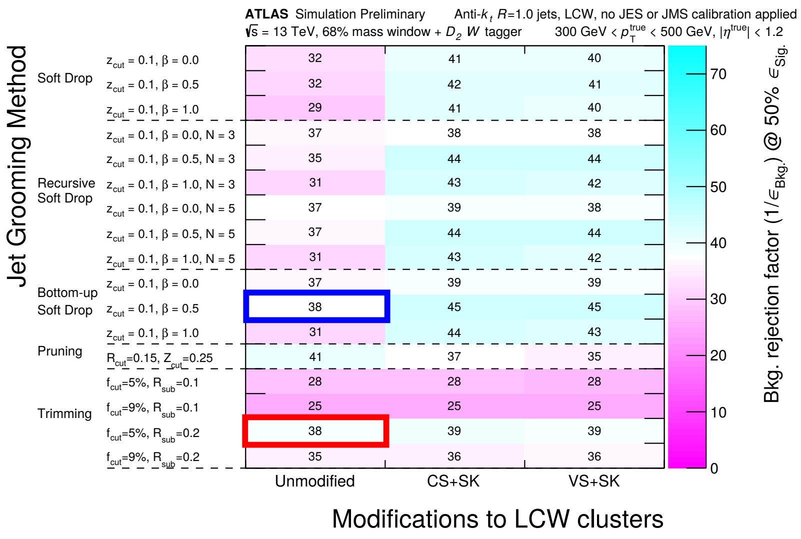
  - We will begin by taking some "classic" choices for these different grooming configurations. For more details on the SoftDrop variants and what the different parameters of each approach mean, you can see the SoftDrop fastjet-contrib page here
  - We can configure these tools as follows:

```
fastjet::Pruner pruner(fastjet::cambridge_algorithm, 0.1, 0.5);
fastjet::contrib::SoftDrop             sd  (  2,0.1,    1.0);
fastjet::contrib::RecursiveSoftDrop rsd  (  2,0.1,-1,1.0);
fastjet::contrib::BottomUpSoftDrop busd (  2,0.1,    1.0);
```

  - While those are "classic" choices for the different grooming algorithms, it is worth noting that they may not be using optimal settings for jets built from topoclusters, as used by ATLAS. This is important, because ATLAS has optimized the trimming parameters they use in order to work well for such jets. As such, we will also consider one additional SoftDrop configuration:

```
fastjet::contrib::BottomUpSoftDrop busdt(0.5,0.1,    1.0);
```

– This configuration (blue box) has been found to have similar W-tagging performance in ATLAS to trimming (red box), according to the following plot (taken from here)

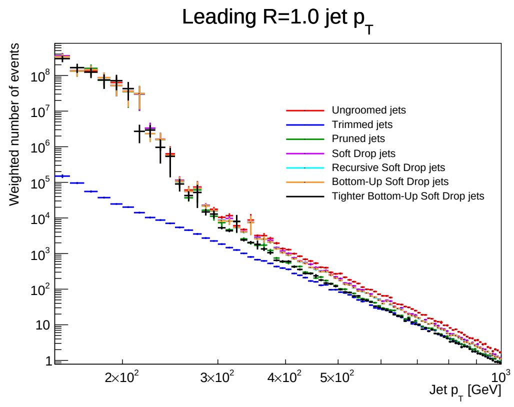


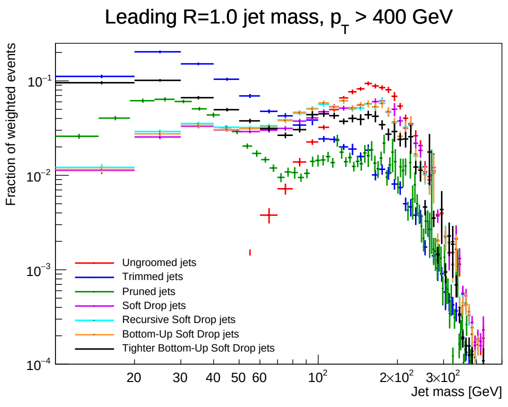
Method	Configuration	Unmodified	CS+SK	VS+SK
Soft Drop	z_{cut} = 0.1, β = 0.0	32	41	40
	z_{cut} = 0.1, β = 0.5	32	42	41
	z_{cut} = 0.1, β = 1.0	29	41	40
Recursive Soft Drop	z_{cut} = 0.1, β = 0.0, N = 3	37	38	38
	z_{cut} = 0.1, β = 0.5, N = 3	35	44	44
	z_{cut} = 0.1, β = 1.0, N = 3	31	43	42
	z_{cut} = 0.1, β = 0.0, N = 5	37	39	38
	z_{cut} = 0.1, β = 0.5, N = 5	37	44	44
	z_{cut} = 0.1, β = 1.0, N = 5	31	43	42
Bottom-up Soft Drop	z_{cut} = 0.1, β = 0.0	37	39	39
	z_{cut} = 0.1, β = 0.5	38	45	45
	z_{cut} = 0.1, β = 1.0	31	44	43
Pruning	R_{cut}=0.15, Z_{cut}=0.25	41	37	35
Trimming	f_{cut}=5%, R_{sub}=0.1	28	28	28
	f_{cut}=9%, R_{sub}=0.1	25	25	25
	f_{cut}=5%, R_{sub}=0.2	38	39	39
	f_{cut}=9%, R_{sub}=0.2	35	36	36

– With the various grooming tools configured, we can apply them to the ungroomed jets that we built in Step 3 and add the leading jet p_{T} and the leading jet mass (if the p_{T} is above 400 GeV) to histograms. Below is the code to do this for the pruning algorithm, and you need to repeat the same code for each of the SoftDrop grooming algorithms we just configured.

```
if (jets_a10_clusters.size())
{
  const fastjet::PseudoJet& ungroomed = jets_a10_clusters.at(0);
  fastjet::PseudoJet pruned    = pruner(ungroomed);

  hist_mypruned_pt.Fill(pruned.pt(),EventWeight);
  if (pruned.pt() > 400.e3)
    hist_mypruned_m.Fill(pruned.m(),EventWeight);
}
```

– The resulting plots comparing these different approaches are as follows:





– These plots show that trimming is really different than the other options for the jet p_{T}, and it also has the least evidence for a mass peak in the ∼200 GeV regime. This is part of why ATLAS has chosen to use this grooming algorithm for so long. However, I have to once again stress that the other grooming algorithms on this plot have not been optimized nearly as much as the trimming algorithm in the context of $R = 1.0$ jets built using ATLAS topoclusters under high pileup conditions, and thus this should not be taken as an absolute statement on the performance of any given algorithm!

– With that done, you now have created plots for a wide variety of grooming algorithms which were not in the input file. That's excellent, as now you can easily change to whatever other grooming algorithms you feel like, or try other grooming parameters, or build jets in different ways. You now have harnessed the power of jet reconstruction and grooming in fastjet.

- **Step 5: Calculating substructure variables for R=1.0 jets**
  - Now that we have built and groomed our jets, there is one last task that we should discuss. Often, we want to calculate jet substructure variables. These typically loop over all of the constituents which a jet is made of, and often the order in which they were grouped together when building the jet, in order to study some angular or energy correlation structure within the jet. Such variables can be very powerful in identifying jets originating from the hadronic decays of massive particles (W, Z, H, or top) and rejecting particles coming from QCD processes (light quarks or gluons).
  - Two very common varibles are $D_2^{\beta=1}$ for W/Z identification and τ_{32}^{WTA} for top-quark identification. The $D_2^{\beta=1}$ variable is a ratio of energy correlation functions, while the τ_{32}^{WTA} variable is a ratio of what is called N-subjettiness. More details on calculating these variables and the different ways they can be configured or defined can be found on their respective fastjet-contrib pages: EnergyCorrelator and Nsubjettiness
  - To calculate these variables, we need two more header files:

```
#include "fastjet/contrib/EnergyCorrelator.hh"
#include "fastjet/contrib/Nsubjettiness.hh"
```

  - We also need to define the base variables which our more advanced variables are calculated from:
    * $D_2 = \mathrm{ECF3} \times (\mathrm{ECF1})^3/(\mathrm{ECF2})^3$
    * $\tau_{32} = \tau_3/\tau_2$

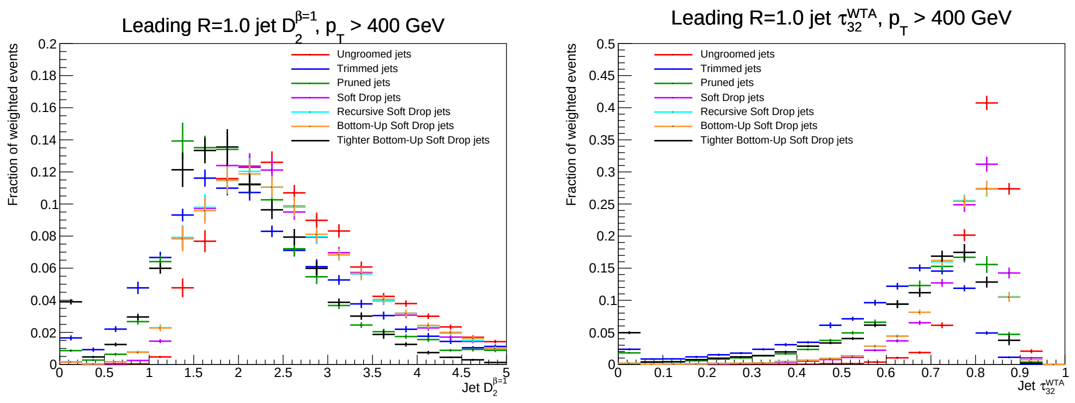
```
fastjet::contrib::EnergyCorrelator ECF1(1, 1.0,
            fastjet::contrib::EnergyCorrelator::pt_R);
fastjet::contrib::EnergyCorrelator ECF2(2, 1.0,
            fastjet::contrib::EnergyCorrelator::pt_R);
fastjet::contrib::EnergyCorrelator ECF3(3, 1.0,
            fastjet::contrib::EnergyCorrelator::pt_R);

fastjet::contrib::Nsubjettiness tau2(2,
            fastjet::contrib::OnePass_WTA_KT_Axes(),
            fastjet::contrib::UnnormalizedMeasure(1.0));
fastjet::contrib::Nsubjettiness tau3(3,
            fastjet::contrib::OnePass_WTA_KT_Axes(),
            fastjet::contrib::UnnormalizedMeasure(1.0));
```

  - With the tools defined, we can use them to calculate the variables we want and to fill them in histograms so long as the jet p_{T} is above 400 GeV (same motivation as for the jet mass histograms).

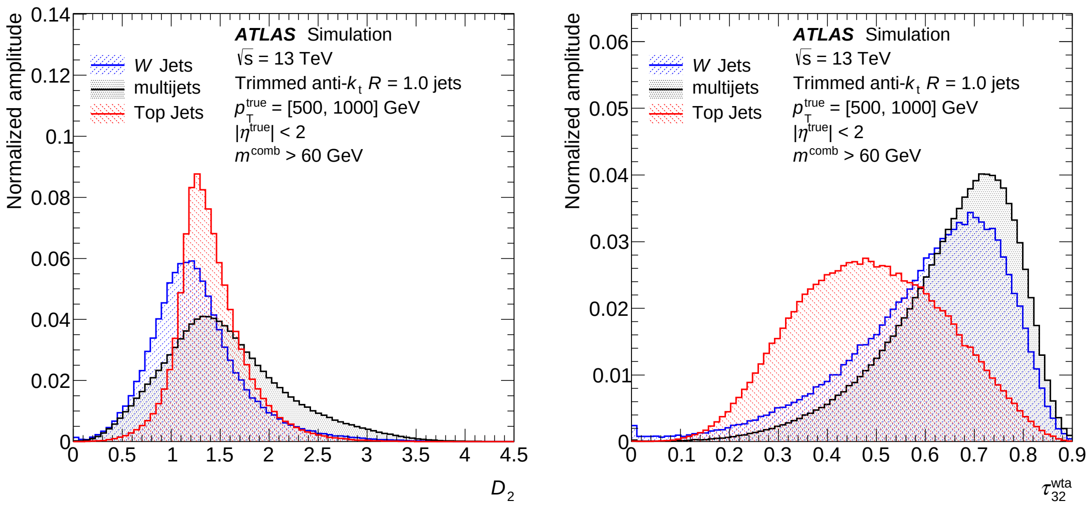
```
if (ungroomed.pt() > 400.e3)
{
  hist_ungroom_D2.Fill(   ECF3(ungroomed)
                        * pow(ECF1(ungroomed),3)
                        / pow(ECF2(ungroomed),3)
                        , EventWeight);
  hist_ungroom_tau32.Fill(   tau3(ungroomed)
                           / tau2(ungroomed)
                           , EventWeight);
}
```

  - Again, this will slow down the code as these variables involve nested loops over all of the clusters that the jets were built from. I would thus recommend only doing this for the leading jet in each event.
  - Note that technically grooming can flip the order of leading vs subleading jets: what was the leading jet before grooming may become subleading after grooming. I would suggest that you ignore this for now, and instead always look at the leading ungroomed jet. If you only groom that jet, and then plot the corresponding p_{T}, mass, D_2, and τ_{32}, then you know you are always comparing the same group of energy deposits in the detector.

- The result for D_2 and τ_{32} of the various grooming algorithms is as follows:



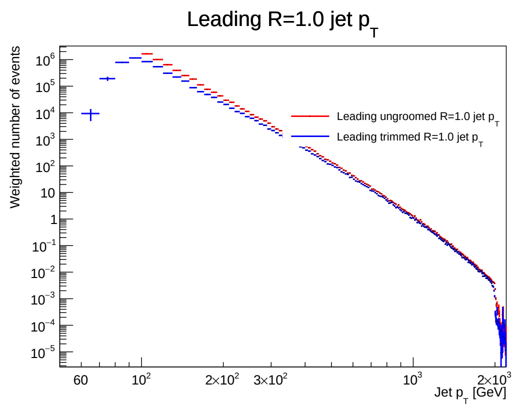
- This is interesting, as while the mass seemed to give clear support to trimming, the substructure variables tell a more complex story. In both plots, the ideal result would have QCD as far to the right as possible and signal (W for D_2 or top for τ_{32}) as far to the left as possible. The motivation for this statement is shown below, as taken from this ATLAS paper. Without looking at how the signal moves for these different grooming algorithms, it's not possible to conclude on which grooming algorithm is the best for these substructure variables.

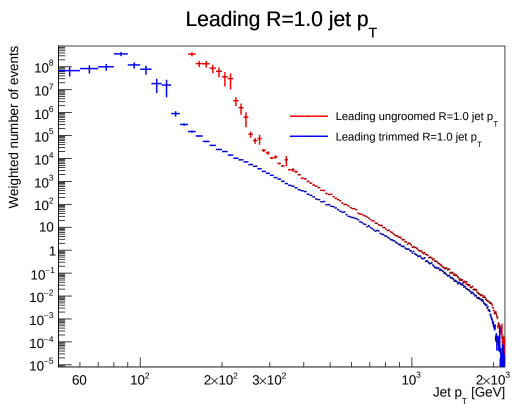


- Good job! You have finished the core part of the exercise, and now you can use fastjet to build jets, groom jets, and calculate substructure variables for jets. If you are interested, there is one last set of comparisons you could make.

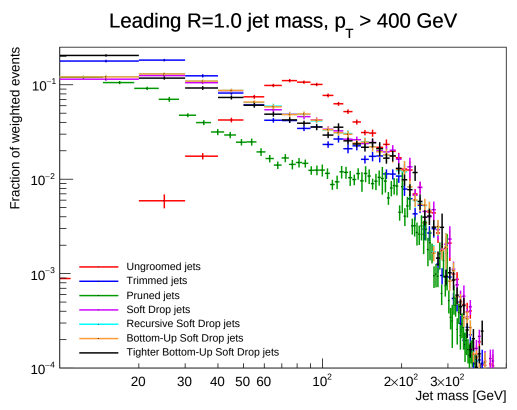
- **Bonus: truth vs reconstructed jets**
  - This last step is not required, and is really more for your interest. If you open the code and search for "bool isTruth", you will find a variable that controls whether the program runs on truth particles (detector-independent) or topoclusters (detector-dependent). If you change this variable to be "true", then you can re-compile and re-run exactly the same code. The outputs will now be for truth jets instead of calorimeter jets. You can then compare topocluster and truth jets to make some interesting observations, as shown on the next page.

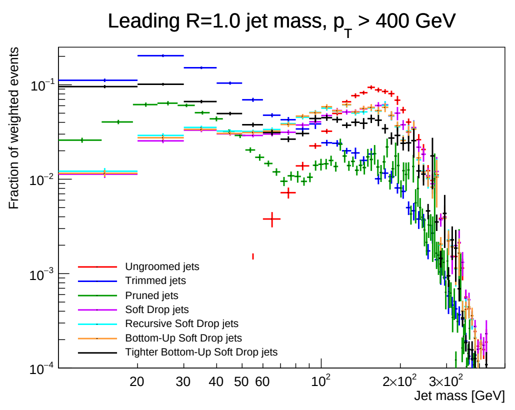
– Pileup vs underlying event, from step 2 (left is truth, right is clusters). This makes it much more clear what I was saying, as truth jets include underlying event contributions but not pileup.



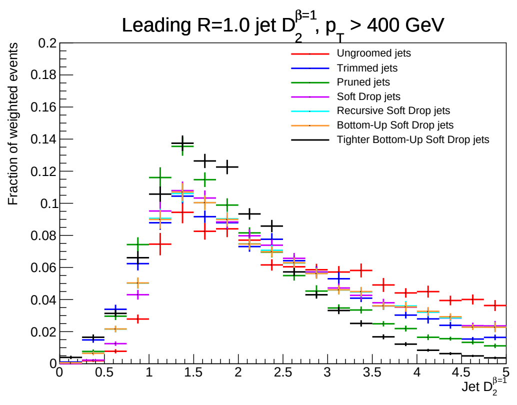


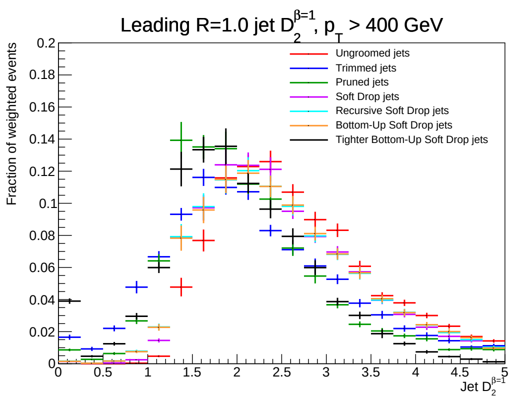
– Jet mass for different grooming choices, from step 4 (left is truth, right is clusters). The difference here is striking - it's clear that many of the grooming algorithms are struggling due to experimental effects, as they generally look more similar at truth level.

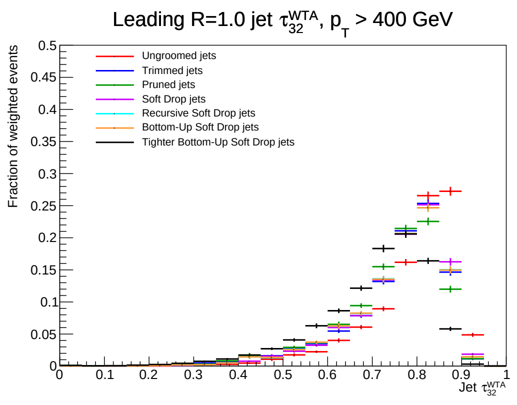


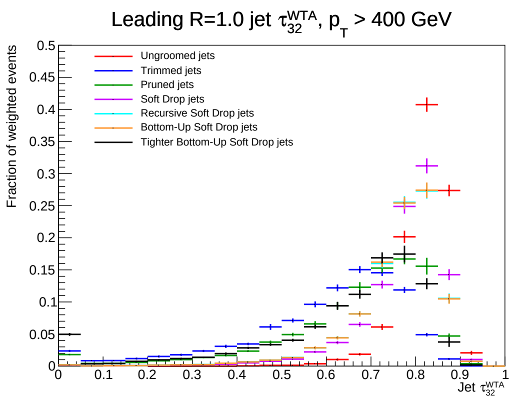


– Jet D_2 and τ_{32}, from step 5 (left is truth, right is clusters). There are also experimental effects here, but it's not as striking as for the jet mass. It would be interesting to see these same plots also with W and top signals overlaid to see if this holds for both the signal and background processes.









- That's it! You have finished the second exercise.