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 GeV, 2000 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)

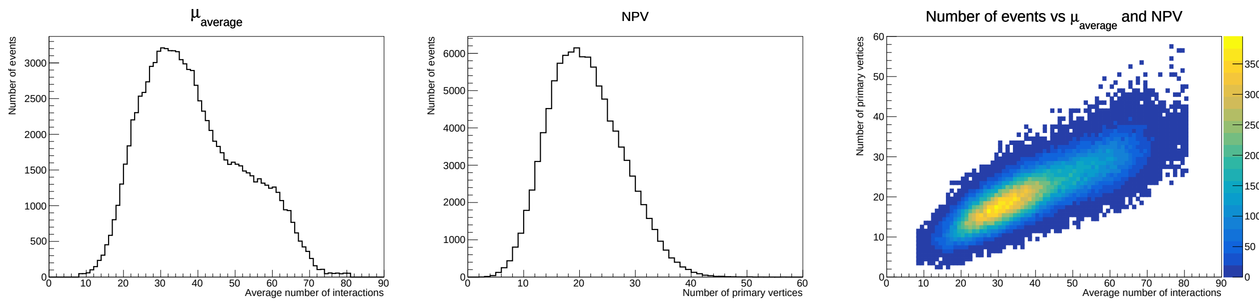
        - The third argument is the name of the tree from the input file, which is `JetRecoTree`
        - The fourth argument is the name of the file that we want to read from and that you downloaded earlier, which by default is named `JetRecoDataset.root`
    - Putting this all together, we get the following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



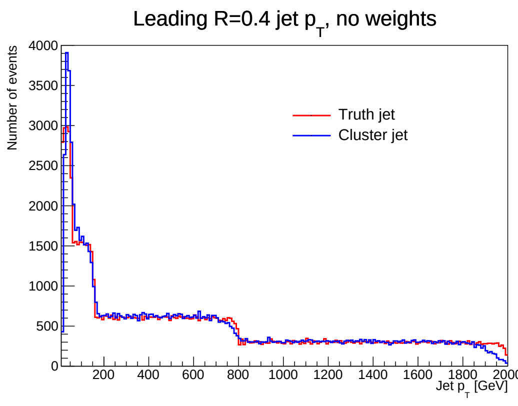
    - This shows the clear correlation between the two pileup measures, but that they are not identical. There is a spread of values, as one is an average quantity over many events while the other is what was observed in a given event. The values of `mu_average` and `NPV` are typically used to represent *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

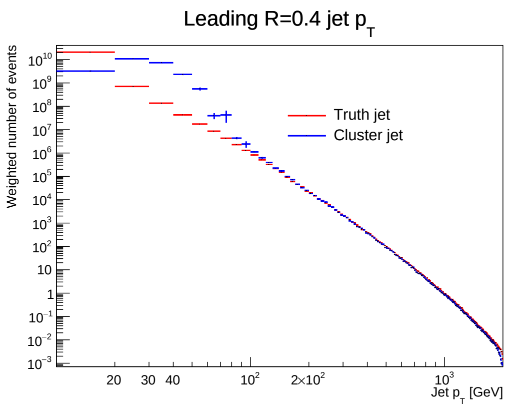
- **Step 2: R=0.4 cluster and truth jets and the event weight**
    - Next, we want to look at the p_T distribution of the leading (highest-p_T) jet. We want to do this for both calorimeter (reconstructed) jets and truth (detector-independent) jets.
    - We want to do this both with and without a so-called `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 $p_{\mathrm{T}} < 100$ 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).

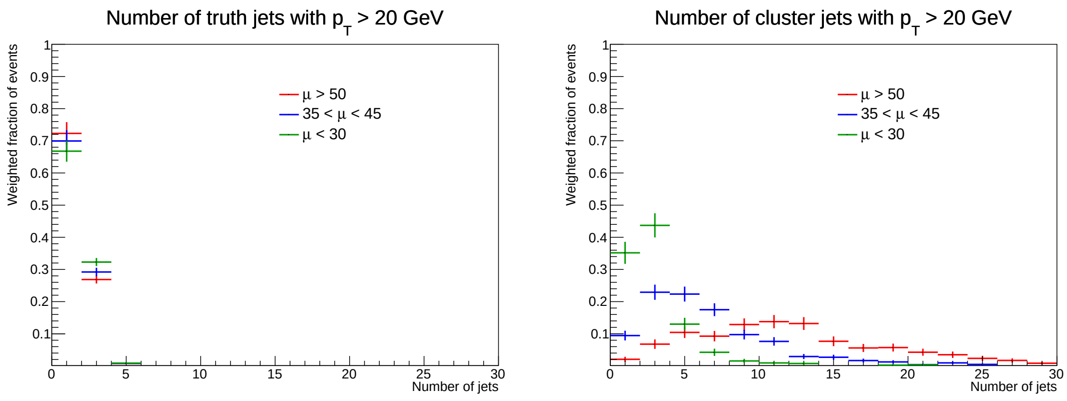
```
// We only want to consider events that have at least one jet
if (numJetReco != 0)
{
  if      (mu_average < 30)
    hist_reco_njets_lowmu.Fill(numJetReco,EventWeight);
  else if (mu_average > 35 && mu_average < 45)
    hist_reco_njets_midmu.Fill(numJetReco,EventWeight);
  else if (mu_average > 50)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

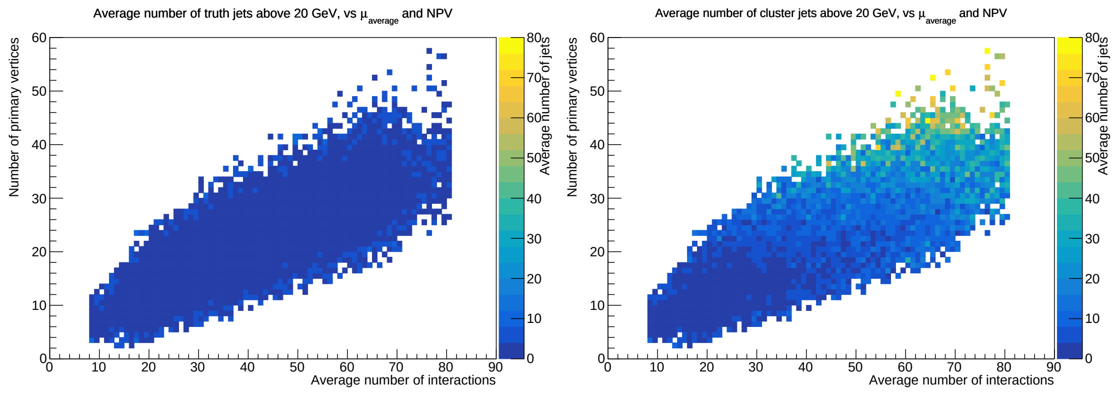
– With that done, with have finished the third step. If you re-compile the code, run it specifying step 3, and also run the plotting script specifying step 3, you should now get an additional four plots. The first two are as follows:



– 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:



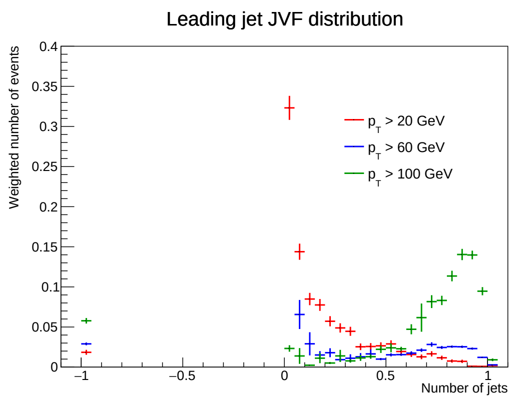
- **Step 4: Tracks and R=0.4 track jets**
  - I mentioned earlier that the calorimeter is sensitive to out-of-time pileup, while the tracker is not as the read-out time is shorter than the time between subsequent beam crossings. Trackers are even more powerful than that, as they can also identify the 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:

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

  - 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);
```

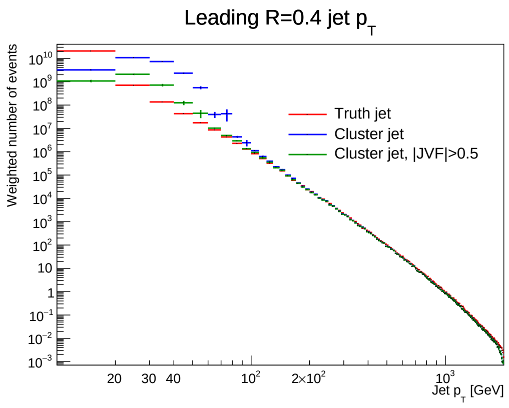
  - The resulting plot shows how the JVF distribution shifts from values of 0 (pileup-like) to 1 (hard-scatter-like) as the p_{T} increases. Note that there is also a population of jets with a JVF value of -1, which corresponds to calorimeter jets that have zero tracks pointing at them. This could be either 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).



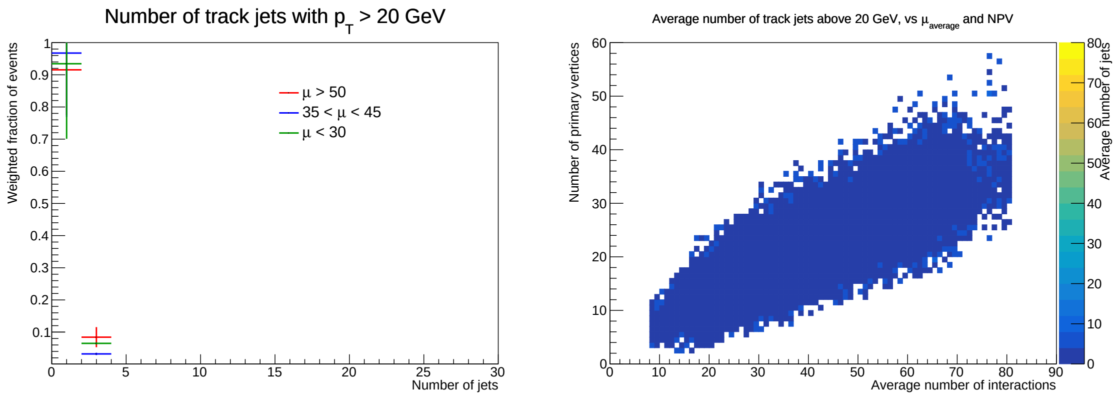
  - A reasonable first value to consider as a JVF cut is 0.5, which means that at least half of the track momentum pointing at a jet comes from the hard-scatter vertex. Note that it is usually better to apply the cut on the absolute value of JVF, or $|\mathrm{JVF}| > 0.5$, in order to not remove jets with 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.



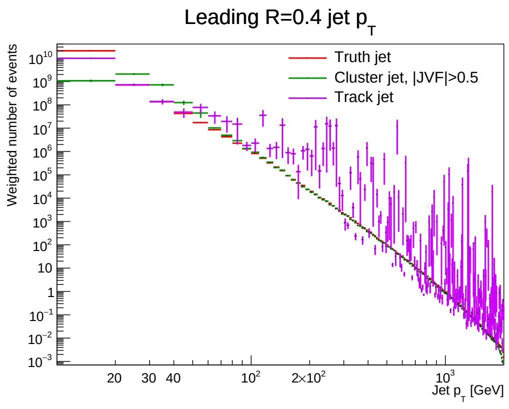
- We have seen how tracks can improve the pileup-robustness of our calorimeter jets. However, what do track jets themselves look like? After all, we can build jets out of any set of four-vectors, so we can build them using tracks only from the hard-scatter vertex that we are actually interested in.
- 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.

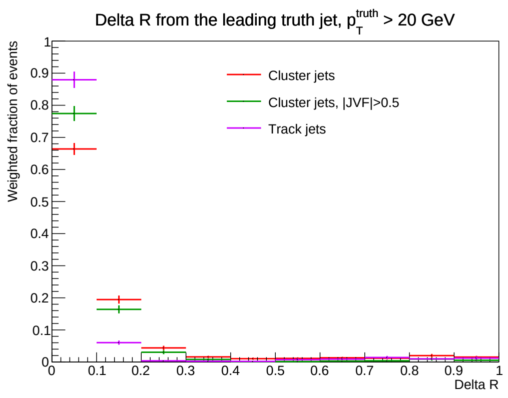
- **Step 5: Jet response studies**
  - In order to better understand what is going on, we need to directly compare truth jets to reconstructed jets (either track jets or calorimeter jets). To do this, we can "match" truth and reconstructed jets, using ΔR (Delta R), the angular distance between the truth 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);
  }
}
```

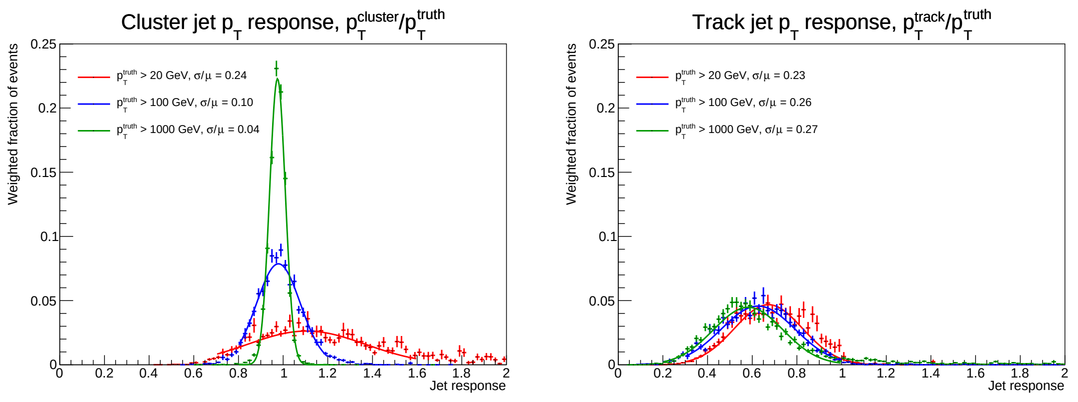
  - With this done for both calorimeter and track jets, the following plot is obtained:



  - 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.