Using pyjetty

Reynier Cruz Torres

May 13, 2021

Contents

1 Introduction

This document gives concrete examples on how to use pyjetty [1], which leverages the package heppy [2] and packages therein. Specifically, I will use the 'jet_axis' analysis as an example. For running other codes, replace the label 'jet_axis' with the appropriate identifier.

2 Processing

2.1 pp

Example on running a local pp data job:

```
python process/user/rey/process_data_energy_drop.py \
-f /rstorage/alice/data/LHC17pq/448/20-06-2020/448_20200619-0610/unmerged/child_1/0001/AnalysisResults.root \
-c config/energy_drop/rey_pp.yaml
```

Example on running a local pp MC job:

```
python process/user/rey/process_mc_energy_drop.py \
-f /rstorage/alice/data/LHC18b8/520/child_1/TrainOutput/1/282008/0001/AnalysisResults.root \
-c config/energy_drop/rey_pp.yaml
```

The measured dataset we are currently using corresponds to LHC17p and LHC17q pass1 AOD. We combine the FAST trigger cluster with the CENT woSDD trigger cluster, both of which are reconstructed without the SDD. Example on running a slurm data job:

```
cd slurm/sbatch/jet_axis/
sbatch slurm_LHC17pq.sh
```

The analysis also uses the LHC18b8 Pythia8 $p_{\mathrm{T,\,hard}}$ MC production (Monash 2013 tune) with the full GEANT3 ALICE detector simulation. The production consists of 20 $p_{\mathrm{T,\,hard}}$ bins, each populated with approximately 6M events, with bin edges: $[5, 7, 9, 12, 16, 21, 28, 36, 45, 57, 70, 85, 99, 115, 132, 150, 65$
GeV/c. The MC is anchored run-by-run to LHC17pq runs. Example on running a slurm MC job:

```
cd slurm/sbatch/jet_axis/
sbatch slurm_LHC18b8.sh
```

2.2 Pb-Pb

The measured dataset we are currently using corresponds to the LHC18q and LHC18r pass3 AOD. Example on running a slurm data job:

```
cd slurm/sbatch/jet_axis/PbPb/
sbatch slurm_LHC18qr.sh
```

The simulated dataset we are currently using corresponds to the LHC20g4 Pythia8 $p_{\mathrm{T,\,hard}}$ MC production (Monash 2013 tune) with the full GEANT3 ALICE detector simulation. The production consists of 20 $p_{\mathrm{T,\,hard}}$ bins, each populated with approximately 8M events, with bin edges: $[5, 7, 9, 12, 16, 21, 28, 36, 45, 57, 70, 85, 99, 115, 132, 150, 101169, 190, 212, 235, 235+]$ GeV/c. Example on running a slurm data job:

```
cd slurm/sbatch/jet_axis/PbPb/
sbatch slurm_LHC20g4_embedding.sh
```

Compared to pp, the Pb-Pb analysis is very heavy. First of all, we have much more Pb-Pb statistics than pp. Additionally, the MC embedding makes the process slower. Thus, for tests we can use 10% of the measured and simulated data. For these, use the following instead of the previous two sets of instructions:

```
cd slurm/sbatch/jet_axis/PbPb/
sbatch slurm_LHC18qr_10percent.sh
```

```
cd slurm/sbatch/jet_axis/PbPb/
sbatch slurm_LHC20g4_embedding_10percent.sh
```

3 Merging data root files

- `cd pyjetty/pyjetty/alice_analysis/slurm/utils/rey`
- open the file `merge_data.sh`

  ```
  #! /bin/bash
  #
  # Script to merge output ROOT files
  JOB_ID=209383
  FILE_DIR="/rstorage/alice/AnalysisResults/rey/$JOB_ID"
  FILES=$( find "$FILE_DIR" -name "*.root" )
  OUTPUT_DIR=/rstorage/alice/AnalysisResults/rey/$JOB_ID
  hadd -f -j 20 $OUTPUT_DIR/AnalysisResultsFinal.root $FILES
  ```

- edit this file and replace `rey` with your username and edit the number in `JOB_ID=209383` with the correct run number that you would like to merge
- `source merge_data.sh`

4 Scaling and merging MC root files

The MC files are produced separate in $\hat{p}_T$ bins. This is done to focus the generation in different bins and accrue similar amount of statistics even in the bins where the cross section is small. Consequently, these files need to be scaled by the cross section before combining them.

1. hadd all root files corresponding to the same $\hat{p}_T$ bin. See, for example: slurm_merge_LHC18b8.sh and merge_LHC18b8.sh. Edit both files and replace `rey` with the appropriate username. Also, modify the number in `JOB_ID=209384` in `merge_LHC18b8.sh` to reflect the right job number.

```
sbatch slurm_merge_LHC18b8.sh
```

2. cd into the directory containing the 1/, 2/, ... sub-directories and scale the combined files corresponding to a given $\hat{p}_T$ bin by the appropriate scale factor. To do so, run scaleHistograms.py (with the correct file path) and config file associated with the simulation, e.g. -c /rstorage/alice/data/LHC18b8/scaleFactors.yaml. Here's an example:

```
python /home/rey/pyjetty/pyjetty/alice_analysis/slurm/utils/rey/scaleHistograms.py \
-c /rstorage/alice/data/LHC18b8/scaleFactors.yaml
```

The path given above is specifically for the PYTHIA8 + GEANT3 simulations. The paths for the fast PYTHIA8 and fast HERWIG7 simulations are:

```
/rstorage/generators/pythia_alice/tree_fastsim/scaleFactors.yaml
/rstorage/generators/herwig_alice/tree_fastsim/scaleFactors.yaml
```

respectively.

In the case of the HERWIG7 fast simulation, we observed some outliers. To clean them, do:

```
python /home/rey/pyjetty/pyjetty/alice_analysis/slurm/utils/rey/scaleHistograms_fastHerwig.py
```

after the previous step.

3. After the histograms have been scaled, you should merge the $\hat{p}_T$ bins. See for example merge_pthat.sh. The number in the line `JOB_ID=209384` and paths should be updated. Then do:

```
source merge_pthat.sh
```

5 Analysis

5.1 Writing analysis code

You need to begin by creating a code in: `pyjetty/pyjetty/alice_analysis/analysis/user/`. This code will inherit from /substructure/run_analysis.py. For an example analysis code see: run_analysis_energy_drop.py.

5.1.1 Functions the user needs to implement

There are three main functions the user needs to implement in the analysis code:

- `plot_single_result()`
- `plot_all_results()`
- `plot_performance()`

The function names are self-explanatory.

You also need to edit two functions in analysis/user/substructure/analysis_utils_obs.py: `formatted_subobs_label` and `prior_scale_factor_obs`.

5.2 Running analysis code

```
python analysis/user/rey/run_analysis_energy_drop.py  -c config/energy_drop/rey_pp.yaml
```

5.3 What happens when you run the analysis code

Right away, what the code does is it runs the ‘main’ function run_analysis() defined in `run_analysis.py`. The first step in this function is to do unfolding (if the user requested it) through the function perform_unfolding(), also defined in `run_analysis.py`.

This function loops over the ‘systematic’ settings defined in the config file. For each setting, the code sets variables related to inputs and outputs:

```
output_dir = getattr(self, ‘output_dir_{}’.format(systematic))
data = self.main_data
response = self.main_response
main_response_location = os.path.join(getattr(self, ‘output_dir_main’), ‘response.root’)
rebin_response = self.check_rebin_response(output_dir)
```

It the initializes some variables:

```
prior_variation_parameter = 0.
truncation = False
binning = False
R_max =  self.R_max
prong_matching_response = False
```

And it finally sets these variables depending on the systematic setting to be unfolded:

```
if systematic == ‘trkeff’:
   response = self.trkeff_response
elif systematic == ‘prior1’:
   prior_variation_parameter = self.prior1_variation_parameter
elif systematic == ‘prior2’:
   prior_variation_parameter = self.prior2_variation_parameter
elif systematic == ‘truncation’:
   truncation = True
elif systematic == ‘binning’:
   binning = True
elif systematic == ‘subtraction1’:
   R_max = self.R_max1
elif systematic == ‘subtraction2’:
   R_max = self.R_max2
elif systematic == ‘prong_matching’:
   prong_matching_response = True
```

Once these variables have been properly set, the code creates an instance of the Roounfold_Obs class, and subsequently runs the function roounfold_obs().

5.3.1 Unfolding

The `Roounfold_Obs` class and the `roounfold_obs()` function are defined in roounfold_obs.py.

When the instance of the `Roounfold_Obs` class is created, the function `create_output_dirs()` is called. This function creates the following directories: ‘RM’, ‘Data’, ‘KinematicEfficiency’, ‘Unfolded_obs’, ‘Unfolded_pt’, ‘Unfolded_ratio’, ‘Unfolded_stat_uncert’, ‘Test_StatisticalClosure’, ‘Test_Refolding’, ‘Correlation_Coefficients’ and if the variable `thermal_model` is true, ‘Test_ThermalClosure’. Also, two directories called ‘Test_ShapeClosure{}’ are created. Here, {} corresponds to the prior variation parameters defined at the bottom of the config file (with periods removed). For instance, if the config file has:

```
prior1_variation_parameter: 0.5
prior2_variation_parameter: -0.5
```

then you will get the directories ‘Test_ShapeClosure-05’ and ‘Test_ShapeClosure05’.

The unfolding procedure is done two-dimensionally in the observable and in p_{T}. The response matrix corresponds to:

$$\Lambda = (p_{\mathrm{T,det}}, p_{\mathrm{T,true}}, \mathrm{observable}_{\mathrm{det}}, \mathrm{observable}_{\mathrm{true}}). \tag{1}$$

This matrix is then used to unfold the data using Bayes' theorem:

$$P(T|O,\Lambda) = \frac{P(O|T,\Lambda) \cdot P(T)}{P(O)}, \tag{2}$$

where $P(T|O,\Lambda)$ is the likelihood of the truth (T) occurring given that the observation (O) is true. Similarly, $P(O|T,\Lambda)$ is the likelihood of O occurring given that T is true. $P(T)$ and $P(O)$ are the marginal probabilities of observing T and O, respectively.

5.3.2 Unfolding tests

During the unfolding procedure, three validation tests are carried out:

1. **Refolding test:** the Response Matrix (RM) is multiplied by the unfolded result, and compared to the original detector-level distribution. This is done in roounfold_obs.py in the `refolding_test` function. Before doing the refolding, the kinematic-efficiency correction is reverted. Then, the output plots are created in `plot_obs_refolded_slice`. In these plots, the folded truth level and the detector-level (i.e. pre-unfolding) data are compared.

2. **Statistical closure test:**

   - MC det-level is smeared by an amount equal to the measured statistical uncertainty
   - the smeared det-level MC is then unfolded
   - unfolded smeared MC is compared to MC truth-level

   This test checks whether the unfolding procedure is insensitive to statistical fluctuations of the measured spectra. This is done in roounfold_obs.py in the `statistical_closure_test` function. Then, the output plots are created in `plot_obs_closure_slice`.

3. **Shape closure test:**

   - MC det-level and MC truth-level spectra are scaled
   - scaled MC det-level spectrum is unfolded
   - MC truth-level and unfolded scaled MC det-level spectra are compared

   This test checks whether the unfolding procedure is insensitive to the shape of the measured distribution. This is done in roounfold_obs.py in the `shape_closure_test` function, which calls the `shape_closure_test_single` function twice, once with each shape-variation parameter. Then, the output plots are created in `plot_obs_closure_slice`.

5.3.3 Kinematic efficiency

The kinematic efficiency is calculated in roounfold_obs.py in the `plot_kinematic_efficiency` function. 1D slices are plotted in `plot_kinematic_efficiency_projections`. In the case of the jet axis analysis, this is defined as:

$$\varepsilon_{\text{kin}}(\Delta R_{\text{true}}, p_{\text{T,true}}) \equiv \frac{\frac{dN}{d\Delta R_{\text{true}}}\left(\Delta R_{\text{det}} \in \left[0, R/2\right], p_{\text{T,det}} \in \left[10, 80\right]\right)}{\frac{dN}{d\Delta R_{\text{true}}}\left(\Delta R_{\text{det}} \in \left[0, R/2\right], p_{\text{T,det}} \in \left[0, \infty\right]\right)} \tag{3}$$

5.4 Systematic Uncertainties

By default, pyjetty extracts the following sources of systematics:

- track efficiency
- regularization parameter
- priors 1, 2
- truncation
- binning

The explanation for each of these systematics is given below.

5.4.1 Track efficiency

The uncertainty on the tracking efficiency is approximately 4% for hybrid tracks [3, 4, 5, 6]. To assign a systematic uncertainty that accounts for this effect, we construct a response matrix by randomly rejecting 4% of tracks in jet finding. The resulting response matrix is then used to unfold the data.

5.4.2 Regularization parameter

5.4.3 Priors 1, 2

5.4.4 Truncation

5.4.5 Binning

6 Generating PYTHIA events within heppy

I will use the jet-axis analysis as an example.
cd into `pyjetty/pyjetty/alice_analysis/slurm/sbatch/jet_axis` and do:

```
sbatch pythia_gen_jet_axis_slurm.sh
```

This shell script is running jobs over the code: `pyjetty/pyjetty/alice_analysis/process/user/rey/pythia_parton_hadron.py` After the jobs are finished, cd into: `/home/rey/pyjetty/pyjetty/alice_analysis/slurm/utils/rey/gen/`. To merge the subjobs for each pT-bins, edit the run number in `merge_pythia.sh`, and then do:

```
sbatch slurm_merge_pythia.sh
```

If people are using the cluster and you cannot wait, then do this last step locally by editing the run number in `local_merge_gen.sh`, and then doing:

```
source sourceme_local_merge_gen.sh
```

Finally, merge the different pT-hat-bin files using the same method described in the last step of section 4. No scaling is needed, since this is already done in the `pythia_parton_hadron.py` code.

References

[1] “pyjetty.” https://github.com/matplo/pyjetty.

[2] “heppy.” https://github.com/matplo/heppy.

[3] “ALICE analysis note, Measurement of charged jet cross section in pp collisions at $\sqrt{s_{NN}} = 5.02$ TeV.” https://alice-notes.web.cern.ch/node/534.

[4] “ALICE analysis note, Measurement of charged jet spectra in Pb-Pb collisions at $\sqrt{s_{NN}} = 5.02$ TeV with ALICE at LHC (update including high interaction Pb-Pb runs).” https://aliceinfo.cern.ch/Notes/node/818.

[5] “James’ analysis note.”

[6] “Ezra’s analysis note.”