Tutorial: Truth-level analysis and multiweight variations using Rivet

Christian Gütschow (UCL)

October, 2024

1 Introduction

In this tutorial we will use Rivet to analyse particle-level events and discuss a few common issues to do with fiducial particle-level definitions. At the end, you will be able to write and run simple analysis routines using Rivet, produce and plot histograms and use on-the-fly weight variations to estimate some of the associated generator-level uncertainties.

2 Environment setup

Unless you have a local Rivet installation that you can use, consider installing the latest Rivet docker container, e.g. using

```
docker pull hepstore/rivet:X.Y.Z
```

In case of problems with disk space, you may wish to consider running `docker container prune` and `docker system prune` first. You can run the container interactively, giving read/write access to your local directory using

```
docker container run -it -v $PWD:$PWD -w $PWD hepstore/rivet:X.Y.Z /bin/bash
```

The tarball with the HepMC events used for this tutorial can be obtained as follows

```
wget "https://rivetval.web.cern.ch/rivetval/TUTORIAL/truth-analysis.tar.gz" -O- | tar -xz --no-same-owner
```

3 Getting started

In order to start a fresh Rivet routine, you can simply use the built-in script:

```
rivet-mkanalysis MY_ROUTINE
```

This will produce a mostly empty routine skeleton (a `.cc` file), along with some auxiliary files, such as a `.plot` file that can later on be used to define plotting cosmetics (e.g. axis labels) or and `.info` which contains meta data about the analysis and is mainly relevant when submitting an official analysis routine.

Alternatively, you can base it on an existing routine. Currently, Rivet comes with $\mathcal{O}(10^3)$ routines that document the analysis logic of particle-physics measurements from various collider setups, beam types and beam energies. Existing routines can be browsed and inspected using

```
rivet --list-analyses
rivet --list-analyses ATLAS_
rivet --show-analysis ATLAS_2019_I1718132
```

where the second command is an example for a more refined search and the final command displays the meta data found in the associated `.info` file for the routine `ATLAS_2019_I1718132`. As you can tell, Rivet knows a lot about its analyses via the associated `.info` file!

For this tutorial, we have prepared an initial draft routine for you already! Let's take a look at the content of the file called `MY_ANALYSIS.cc`, printed in full on the next page[1].

[1] The materials for this tutorial can also be found in the Rivet repo.

```
#include "Rivet/Analysis.hh"
#include "Rivet/Projections/FinalState.hh"
#include "Rivet/Projections/FastJets.hh"

namespace Rivet {

  /// @brief Add a short analysis description here
  class MY_ANALYSIS : public Analysis {
  public:

    /// Constructor
    RIVET_DEFAULT_ANALYSIS_CTOR(MY_ANALYSIS);

    /// @name Analysis methods
    /// @{

    /// Book histograms and initialise projections before the run
    void init() {

      _lmode = 0; // default accepts either channel
      if ( getOption("LMODE") == "EL" )  _lmode = 1;
      if ( getOption("LMODE") == "MU" )  _lmode = 2;

      // Book histograms
      vector<double> mll_bins = { 66., 74., 78., 82., 84., 86., 88., 89., 90., 91.,
                                  92., 93., 94., 96., 98., 100., 104., 108., 116. };
      book(_h["mll"], "mass_ll", mll_bins);
      //book(_h["HT"],         "HT",          6,  20., 110.);
      //book(_h["pTmiss"],     "pTmiss",     10,   0., 100.);

      //book(_d["jets_excl"],  "jets_excl",  {0, 1, 2, 3, 4, 5});
      //book(_d["bjets_excl"], "bjets_excl", {0, 1, 2});
    }

    /// Perform the per-event analysis
    void analyze(const Event& event) {

      /// Todo: Reconstruct the dilepton invariant mass to fill the histogram
      // ...
      _h["mll"]->fill(1.0);

    }

    /// Normalise histograms etc., after the run
    void finalize() {

      const double sf = crossSection() / sumOfWeights();
      scale(_h, sf);
      scale(_d, sf);

    }

    /// @}

    /// @name Histograms
    /// @{
    map<string, Histo1DPtr> _h;
    map<string, BinnedHistoPtr<int>> _d;
    size_t _lmode;
    /// @}

  };


  // The hook for the plugin system
  RIVET_DECLARE_PLUGIN(MY_ANALYSIS);
}
```

The routine skeleton has a tyical C++ layout with a few header files at the top, an `init()` method that only runs once at the beginning of the run to initialise the routine, an `analyze(const Event& event)` method that is executed for every single event, a `finalize()` method that is only called once at the end of the routine to do some useful post-analysis opertions (e.g. scaling histograms to cross-section), and finally a couple of member variables towards the end of the routine.

This is can be compiled like any old C++ library, and Rivet provides a wrapper script that will add all the relevant compiler flags for you:

```
rivet-build RivetMY_ANALYSIS.so MY_ANALYSIS.cc
```

where `Rivet*.so` is the canonical form of a compiled Rivet plugin. Provided that there are no namespace clashes, you could in principle compile several routines into a single shared object library like so

```
rivet-build RivetUBER.so ROUTINE1.cc ROUTINE2.cc ...
```

How neat is that? Now, in order to run the routine on an input `HepMC` file, you can use

```
rivet --pwd -a MY_ANALYSIS input.hepmc.gz
```

which uses the default Python script to call the Rivet libraries and tell it to run the routine `MY_ANALYSIS` over the file `input.hepmc.gz`. The first argument is to tell Rivet to also look in the present working directory. This is necessary in this case, since Rivet would otherwise not know where to look for a user-supplied routine. If you're very organised and keep your routines in a separate directory, you can also tell Rivet where to look via the environment variable `RIVET_ANALYSIS_PATH`. In our case, defining `export RIVET_ANALYSIS_PATH=$PWD` is equivalent to using the `--pwd` flag.

Congratulations – these are the very basics of running a Rivet routine. Our example routine is not all that useful yet, so let us take another look at the `MY_ANALYSIS.cc` file to better understand what it does.

The first thing that you can see in the `init()` method is a bit of logic to set the value of the member variabe `_lmode`. This is an optional feature that will become useful to steer the analysis logic from the outside using Rivet's options mechanism. The value of `_lmode` defaults to `0` but could also be `1` or `2`, depending on what the option `"LMODE"` (short for 'lepton mode', but could be anything really) is set to when calling the routine. The default value of an option is the empty string (`""`), unless the user specifies a value when calling the routine, e.g. like so

```
rivet -a MY_ANALYSIS:LMODE=EL,MY_ANALYSIS:LMODE=MU input.hepmc.gz
```

which would run two instances of the routine, once using the option using `LMODE=EL` and once with `LMODE=MU`. Any `string`-type value is acceptable, unless a certain set of allowed values is specified in the `.info` file.

The last part of the `init()` method is used to book histograms. The one-dimensional continuous histogram pointer type is called `Histo1DPtr`. Seeing as a routine can quickly accumulate many dozens or even hundreds of histograms, it is often convenient to collect them in a key–value map, e.g. a `map<string,Histo1DPtr>`. This is the underlying C++ type of the object `_h` which is declared near the bottom of the routine. To book a 1D histogram, simply call one of the following

```
book(_h["name1"], "name1", bin_edges);
book(_h["name2"], "name2", nBins, min_edge, max_edge);
book(_h["name3"], "name3", logspace(nBins, min_edge, max_edge));
book(_h["name4"], 1, 2, 3);
```

The first argument is the intended target variable for the booked histogram. In this case we supply a unique key to the map which in turn creates a placeholder. The second argument is the intended histogram name that will be used when writing the histogram to file. This could be the same as the map key, but does not need to be. The remaining arguments are to define the bin edges – you can specify a vector of bin edges directly, or you can supply a number of bins between a minimum bin edge and a maximum bin edge and let Rivet divide evenly across that range. In case the routine comes with a reference-data file, as is often the case for measurements, the syntax shown in the last line can be used

to let Rivet work out the binning based on the reference data. The integers are referring to the integers in the canonical HEPData identifier (e.g. `d01-x02-y03`).

Strictly speaking, `Histo1DPtr` is just an alias for the more generic `BinnedHistoPtr<double>` where `double` is the C++ template parameter used to specify the edge type along the continuous axis. For histograms along discrete axes, such as multiplicty distributions or arbitrary `string` labels, it might be preferable to book `BinnedHistoPtr<int>` or `BinnedHistoPtr<string>` instead, respectively. This also works in higher (axis-)dimensions by adding more type labels as template arguments (e.g. `BinnedHistoPtr<double,int>`).

The `analyze(const Event& event)` function is executed once per event. Currently, all it does is fill a histogram with the number `1.0`[2], which isn't all that useful. It will be up to you to make it do something clever in the next section, but let's first look at the final part of the skeleton.

In the `finalize()` method, the histogram is then just scaled by a factor $\sigma / \sum_i w_i$ where σ is the generator cross-section and $\sum_i w_i$ is the sum of weights. Rivet will try to extract the generator cross-section from the HepMC file, but the value can always be overwritten on the command line by passing the `--cross-section` flag (or more conveniently `--x`). Rivet also keeps track of the total sum of event weights in the sample behind the scenes, so you do not have to worry about implementing a counter to do the boring book-keeping yourself.

Note that the `scale` function can be called on an individual histogram but also an entire vector of histograms or a map with histograms as the value. Of course a `normalize` function exists as well.

4 Reconstructing resonances

The aim of this section is to reconstruct a Z-boson resonance from the generic Z+jets events supplied in one of the HepMC file. In general, the reconstruction of observables should be based on the final-state particles in the event record. The event record can be very complicated, depending on the process, and the details of the implementation usually depend on the Monte Carlo event generator. The good news is that there is no need to navigate the event record since Rivet provides a powerful mechanism for projecting out the result of a final-state-based calculation from the event record: the *projection*. A projection is similar to a filter in a sense – you can specify kinematic requirements on the particles that should be considered in the calculation and Rivet will use this calculation tool to project out the collection of final-state particles that satisfy these constraints from the event record. Projections are *declared* in the `init` method and *applied* in the `analyze` method. Sound a bit abstract? Let's look at an example.

The simplest projection is called `FinalState` and as the name suggests it provides access to all final-state particles. It is already enabled with a header file in our skeleton. It can be declared in the `init` method as follows

```
FinalState fs;
declare(fs, "my_first_projection");
```

and then applied for every event in the `analyze` method like so

```
const FinalState& p1 = apply<FinalState>(event, "my_first_projection");
const Particles& fs_particles = p1.particlesByPt();
```

The first line takes the `event` argument and applies the `FinalState` to it. The result of the calculation is saved as `p1` using a `const` reference for computational efficieny. It's possible to ask questions about the projected result, e.g. in this case we are using it to retrieve a set of all final-state particles ordered by their transverse momentum (hard to soft), which returns a vector of `Particle` objects. The `Particle` class is Rivet's way of combinbing particle identification (e.g. `p.charge()` or `p.pid()` for the PDG ID) and `FourMomentum` (via `p.momentum()` or even just `p.mom()`) information into one class. In fact, the latter is only really needed when wanting to combine different four-momenta, seeing as common kinematic quantities are also directly accessible via standard methods like `pT`, `eta`, `rapidity`, `E`, `phi` as well as many others, including covenient short-hand methods such as `abseta` and `absrap`.

[2]The type of this argument should match the axis type specified in the histogram constructor.

Admittedly, the `FinalState` by itself is a bit dull. You might not even care about the full final state. Let's imagine you are prototyping an analysis for a detector that consists of a tracker but does not have a calorimeter. Such a detector can reconstruct the kinematics of charged particles in a magnetic field, e.g. by tracing out the silicon hits produced by charged pions in the detector, but it would struggle to detect neutral particles. Now, we could loop over the particles we retrieved from the `FinalState` and check the charge of each particle to get the relevant subset. Or we just use the `ChargedFinalState` projection by declaring

```
declare(ChargedFinalState(), "my_second_projection");
```

in the `init` method and using

```
const Particles& cfs_particles = apply<ChargedFinalState>(event, "my_second_projection").particles();
```

which lets the projection deal with the fiddly bits, so that we can concentrate on the high-level physics!

Rivet has a large suite of different projections available, such as the `VisibleFinalState` (clue is in the name), the `PromptFinalState` (all final-state particles that did not originate from a hadron decay) or the `VetoedFinalState` (anything but certain types of particles), just to name a few.

Why should I care, you're asking? Well, for starters projections can take other projections as constructor arguments to augment them, but they become a right treat when supplying a `Cuts` argument. This feature is based on C++11 functors, making for rather expressive argument logic:

```
FinalState charged_tracks(Cuts::charge != 0);
FinalState IDtrack(Cuts::charge != 0 && Cuts::abseta < 2.5);
FinalState allMuons(Cuts::abspid == PID::MUON && Cuts::pT > 20*GeV);
PromptFinalState promptMuons(allMuons);
```

The first line is just an alternative way of specifying the `ChargedFinalState`. The second line asks for the same thing but with an additional requirement on the charged-particle pseudorapidity. The third line is a concise way to ask for all muons with a transverse momentum of at least 20 GeV, and the last line uses the previous result to get the subset of prompt muons, i.e. those not originating from a hadron decay. Imagine having to write all this up using `for` loops. That's why you should care.

The advantage of projections is really threefold:

- They can be supplied with a `Cuts` argument to pull a fairly laborious bit of analysis logic into a neat one-liner.
- Projections can be chained together to construct even more complex final-state subsets.
- When running many routines, Rivet will automatically cache the result of equivalent projections behind the scenes for that extra bit of efficiency.

Why not take a moment to play around with this yourself by having a go at the following exercise?

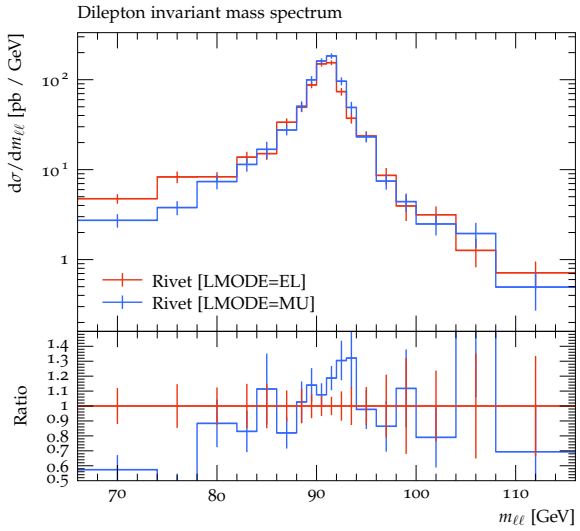
Exercise 1
Modify the provided `.cc` file to select events with exactly one same-flavour opposite-charge lepton pair using leptons with a transverse momentum of at least 10 GeV and restricted to lie within 2.5 in pseudorapdity. Be sure to select an electron or muon pair depending on the value of `_lmode`. Use the selected lepton pairs to reconstruct the dilepton invariant mass and pass it into the prepared histogram. Compile the routine and run it over the provided HepMC events using both lepton-flavour options in the same run.

At the end of the exercise you should have produced a single output file in the `yoda` format. This format is ASCII-based and can be inspected with your favourite text editor. Can you find your filled histogram?

In fact, you should see loads of them! There will be a version for each of the 18 variation weights (including the nominal) in the setup as well as a version before and after running the `finalize` method (watch out for the key word `RAW`) as well as a version for both of the `LMODE` options. We will come back to the variation weights in a later exercise. For now, let us produce a plot comparing the prediction for the electron channel and the muon channel. Rivet provides a nifty little script for you that will make the plots and arrange them in a little HTML-based booklet for you to enjoy them using your favourite browser. You can try this out using the command

```
rivet-mkhtml Rivet.yoda
```

This will create a directory `rivet-plots` which contains an `index.html` file among other things. Check it out!



You should find a Z-boson resonance similar to the plot above. In case you are missing the axis labels, try the

```
export RIVET_ANALYSIS_PATH=$PWD
```

trick to let Rivet know about where to look for the `plot` file with the cosmetic settings that we have prepared for you. Alternatively, you can pass this file directly to the plotting command using the `-c` flag. Does the ratio between the two predictions match your expectation?

5 Bare level vs dressed level

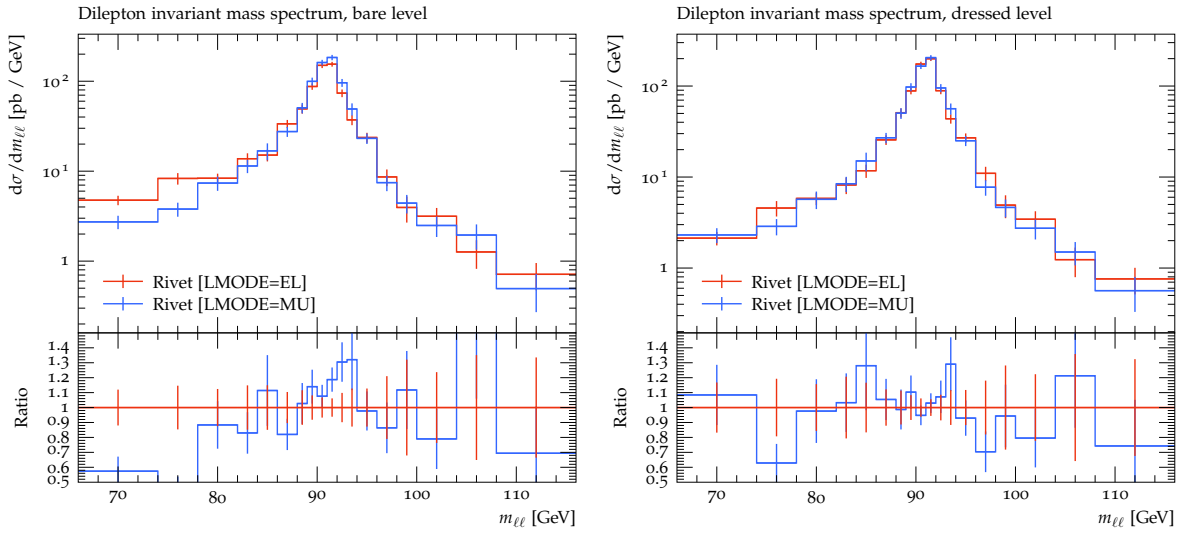
It turns out leptons lose energy through photon radiation and the effect is much more pronounced for electrons due to their lower invariant mass. What we have plotted is the invariant mass of the dilepton spectrum constructed from leptons at the so-called *bare level*, i.e. after photon radiation, which explains why the electron-pair lineshape deviates from the muon-pair version. Luckily, Rivet offers more complex projections that implement experimental strategies, e.g. for lepton dressing, heavy-flavour tagging or mass-windowing and so on. For instance, the `LeptonFinder` projection can be used to dress the bare lepton four-momentum with all photon four-momenta, within a given cone size (typically 0.1) in order to recover energy losses from QED final-state radiation (FSR). Leptons whose four-momenta have been corrected in this way are referred to as being reconstructed at the *dressed level*. The `LeptonFinder` constructor takes the following arguments:

```
LeptonFinder leps(bare-lepton FS, photon FS, dressing-cone size, optional selection cuts);
```

This is the recommended procedure to define leptons at the particle level, since it is more robust against generator differences in the modelling of QED FSR. Let's see this in action!

Exercise 2
Implement an *additional* plot of the dressed-level dilepton invariant mass spectrum in the `cc` file, with otherwise equivalent kinematic requirements on the leptons. Re-compile, re-run and re-plot to check the effect.

At the end of the exercise you should be able to produce two plots similar to the ones on the next page. The one on the left-hand side shows the dilepton invariant mass spectrum constructed from leptons at the bare level, while the one on the right-hand side shows the same spectrum constructed from leptons at the dressed level. The compensating effect of the dressing procedure should be clearly visible when comparing the two ratios.



6 Jet-based observables

Rivet has a lot of neat and flexible built-in support to make your life easier and in this section we will look into how to make use of the most common features. We will start by constructing our own jet collection. Some general advice upfront: avoid coding up built-in methods from scratch as this can be highly error prone! Besides, why reinvent the wheel?

Jets are typically constructed with the `FastJet` program. Rivet has a `FastJets` projection which uses the `FastJet` program internally. The constructor could look like this:

```
FastJets jets(input FS, JetAlg::KT, 1.0, JetMuons::NONE, JetInvisibles::NONE);
```

The first argument is the `FinalState`-based input collection of particles to be clustered. The second and third argument specify the clustering algorithm and associated jet-radius parameter. The last two arguments are optional and can be used to reflect experimental strategies for how to deal with muons or invisibles particles in the clustering. Possible values are `NONE`, `DECAY`, `ALL`; the default being to include all muons (`JetMuons::ALL`) but to exclude all invisible particles (`JetInvisibles::NONE`). Similar to the `particlesByPt()` method we encountered earlier, the `FastJet` projection comes with an equivalent `jetsByPt()` method. It takes an optional `Cuts` argument as well, which allows to efficiently select a subset of the jets that pass certain fiducial requirements. Rivet supports auto-conversion to and from `fastjet::pseudoJet` as well as its own `Jet` class, i.e. it also does not matter which version is passed into the various built-in functions that Rivet has to offer. As always, remember to add the relevant include statement

```
#include "Rivet/Projections/FastJets.hh"
```

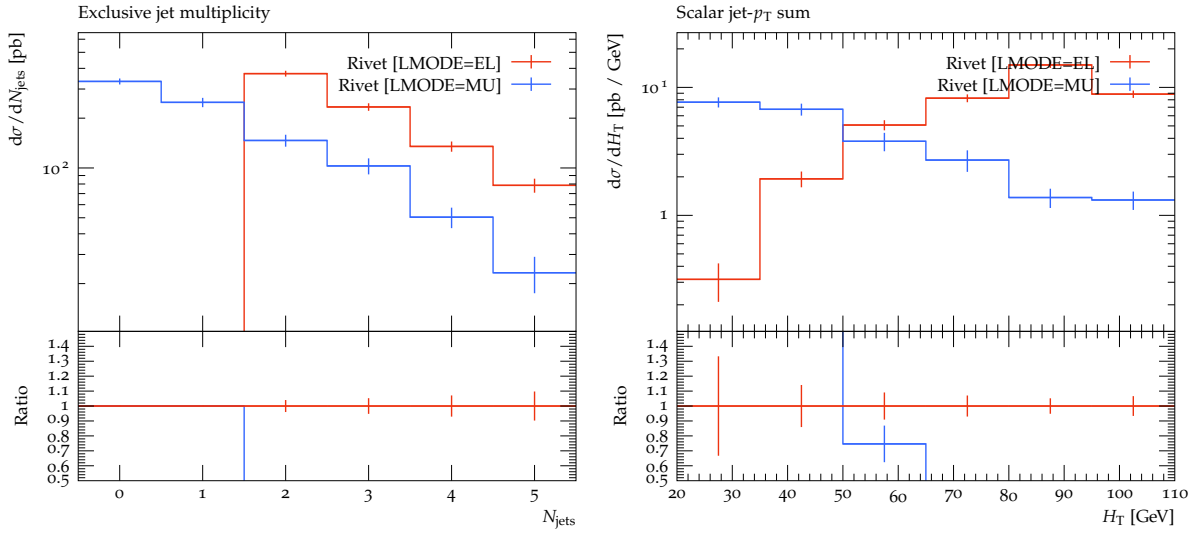
near the top of the routine if it is not already included.

Exercise 3
Select events with 66 GeV $< m_{\ell\ell} <$ 116 GeV at using leptons defined at the dressed level. Using the anti-k_T algorithm with a jet-radius parameter of 0.4, cluster all final-state particles within $|\eta| < 4.9$, except for muons and invisible particles. Select all jets with $p_T > 10$ GeV and within $|y| < 4.5$, then count them to plot the exclusive jet multiplicity. Additionally, add a plot of the scalar jet p_T sum. The latter is often referred to as H_T and is a measure for the amount of hadronic activity in the event. Run this setup separately for each lepton channel and compare the curves. *Optional:* You may find Rivet's built-in `sum` function helpful which takes three arguments:

```
sum(iterable container, property, initial value)
```

and returns a scalar or vector depending on the property called on the elements in the container.

At the end of this exercise, you be able to produce two plots that look something like this:



The differences between the electron and the muon channel are rather striking, but expected considering what goes into the jet clustering. Out of all the visible particles within the pseudorapidity acceptence, only the muons have been removed while the electrons are included into the jet-clustering algorithm as though they were jets. In some sense this is very similar to how electrons and hadrons would leave an energy deposit inside a calorimeter, while muons tend to pass through it and would need to be detected via a dedicated tracker.

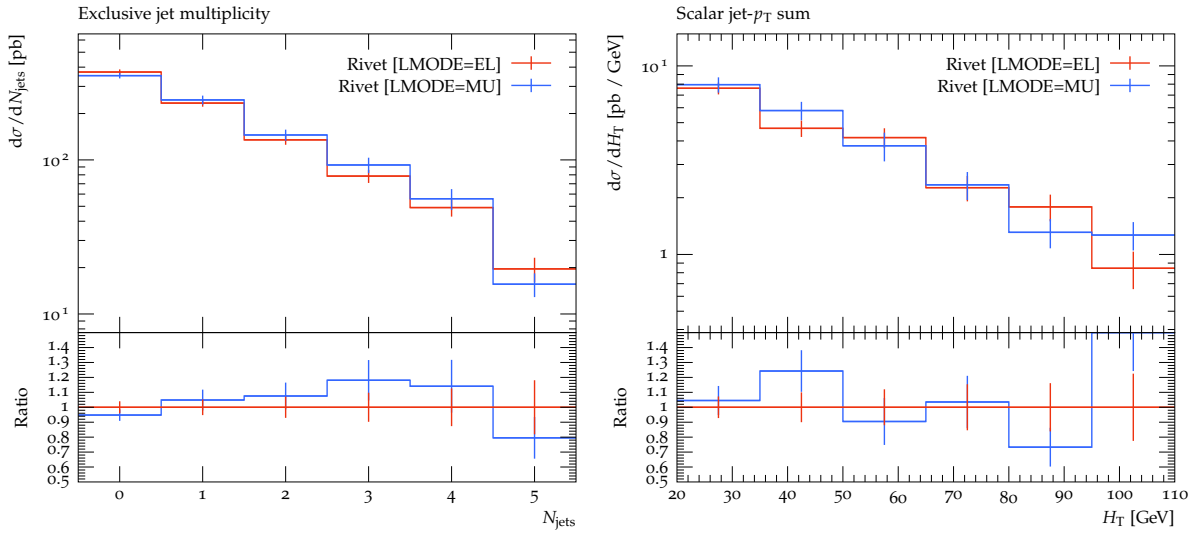
At the particle level one has full flexibility about what to include in the clustering and what not. From the point of view of e.g. ATLAS and CMS, however, usually everything is a jet to start with: Most visible particles produced in an event eventually hit the calorimeter and detector-level jets can be constructed from the energy depositions in the calorimeter cells. Additional dedicated sub-detectors may exist to identify e.g. electron, photon or muon candidates and as a result, a given physics object can appear in several of the object collections reconstructed using the various detector components. This is why "overlap removal" is a thing in experimental analyses. In its simplest form, the objects from the dedicated sub-detectors are preferred, e.g. due to superior resolution or selection efficiency, and so the candidates from other object collections (reconstructed with less precise detector components) which fall into the same fiducial phase space are simply removed from the final state in those object collections.

Exercise 4
Modify the `cc` file to remove all jets that are within $\Delta R < 0.4$ of a selected dressed-level electron or a dressed-level muon, then re-compile, re-run, re-plot and re-compare. *Optional:* You may find Rivet's built-in `discardIfAnyDeltaRLess` function helpful which takes three arguments:

```
discardIfAnyDeltaRLess(input container, reference container, cone size)
```

which returns a copy of the input container with all elements removed that fall within a cone of the specified size with any of the elements of the reference container. An `i` added in front of the function name will modify the input container in place without making a new copy.

At the end of this exercise you should be able to obtain plots similar to those shown on the next page, where the kinematic distributions are now compatible between the electron and the muon channel. Of course you are free to code up the overlap-removal prescription using a standard C++ for loop, but notice how much easier to read the code becomes when pushing the boring bits of experimental strategy into Rivet's built-in functions to leave more headspace for physics. There is another common experimental strategy that emphasises this point even better, which we will focus on in the next section.



7 Heavy-flavour tagging

The preferred way to define b-jets or c-jets at the particle level is to ghost-associate heavy-flavour hadrons to the jets. The idea behind this strategy is to "let the clustering algorithm decide" this in an infrared-safe way. Heavy-flavour hadrons are not stable and hence not included in the jet clustering, which is based on stable final-state particles. "Ghost association" is a technique whereby these hadrons are manually added to the collection of input particles that are to be clustered, but with their 4-momenta scaled down to (effectively) meaningless size, such that the anti-k_T algorithm can just pull them into a jet as it sees fit.

No need to stress: Rivet automatically implements this for you behind the scenes! 'Is my jet a b-jet?' becomes as simple as

```
if (myJet.bTagged()) ...
```

One can even use the familiar `Cuts` argument to refine the tagging further:

```
if (myJet.bTagged(Cuts::abseta < 2.5)) ...
```

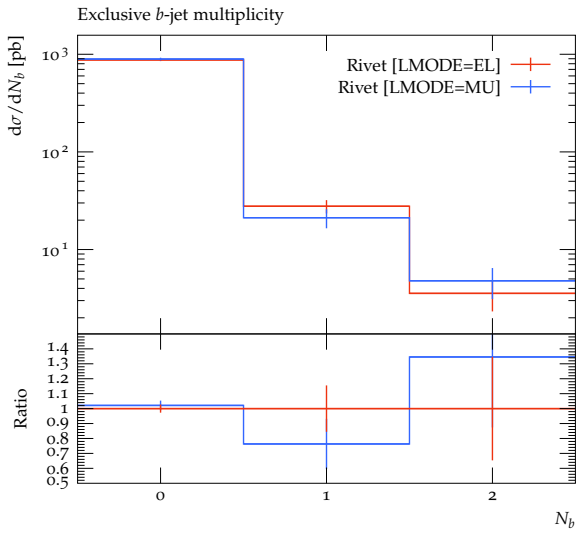
Access to the tagged hadrons is also provided via `myJet.bTags()` and needless to say that similar functions for c-tagging are available as well. Let's practice this!

Exercise 5
Modify the `cc` file to add a plot of the exclusive b-jet multiplicity. *Optional:* You may find Rivet's built-in `count` function helpful which takes up to two arguments

```
count(input container, selection criterion)
```

where the selection criterion could be a `Cuts` argument or a C++ function that is to be called on the elements in the input container, such as Rivet's built-in `hasBTag(Cuts)` method.

At the end of this exercise, you should have produced a plot similar to the one shown on the next page. Notice how much more readible the code becomes when relying on the built-in functions. A typical implementation of ghost-association can easily take somewhere between 200-300 lines of C++ and so re-inventing the wheel here would not only clutter the routine but also boost the chances for bugs to be introduced.

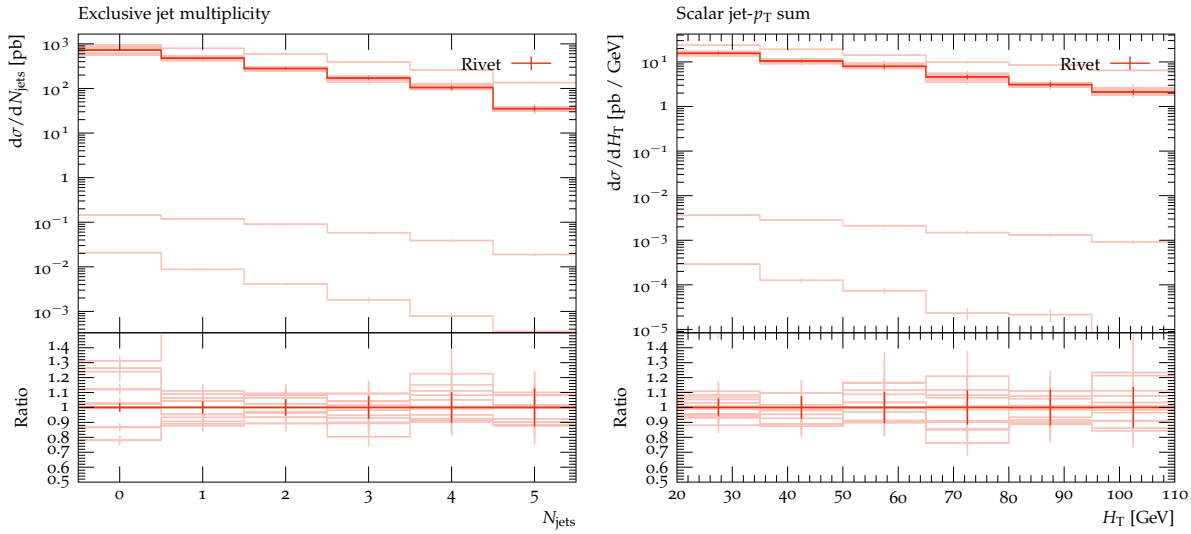


8 Scale uncertainties using on-the-fly variation weights

The events provided for you in the HepMC file were produced using on-the-fly variations of the factorisation and renormalisation scale in matrix element and parton shower. In fact, the setup also includes these variations for the case where only the matrix element scales were varied but not the scales in the parton shower. In this exercise we want to use the variation weights to estimate the corresponding scale uncertainties and add some uncertainty bands to our plot.

Fortunately, we already have all the ingredients to do this, since Rivet will automatically book one histogram per weight variation for you behind the scenes. Out-of-the-box multiweight support is one of the main new features of the Rivet 3 release series! In order to make better use of the available statistics, we recommend to re-run the routine *without* an `LMODE` option, so as to accept either lepton-flavour channel in the same run. Consider running one of the routines included natively in the Rivet release to be able to compare against data as well, such as `ATLAS_2017_I1514251`, the 13 TeV Z+jets measurement (arXiv:1702.05725) from the ATLAS experiment for instance.

Try plotting this with the `--with-variations` flag to the `rivet-mkhtml` command. What you should see is something like the following:



The `--with-variations` flag superimposes the available variation weights in a lighter shading along with the central value. This already gives an idea of the spread of the variations, but it also highlights that there seem to be some variation weights in the setup that correspond to some auxiliary quantity which isn't mean to be used for histogramming. These are the two curves which lie at least a factor $\mathcal{O}(10^4)$ below the central value. Although there is an in-principle agreement on a standard for how to label weights[3], not all generators have yet implemented this standard for their latest public release. While Rivet default behaviour is to be as inclusive as feasible when it comes to weight variations, it also provides some functionality to filter out a specific subset of the weights. In the following we will look at a few examples for how to deal with weights when using the plotting scripts, but in principle similar options are also available to (de-)select weights already at run time (cf. `rivet --help` for the available flags).

Let's take a look at what sort of variation weights are included in the setup:

```
import rivet, yoda
print( set([ rivet.extractWeightName(name) for name in yoda.read('Rivet.yoda') ]) )
```

Here we are using the Python API of YODA to load the output file, loop over all the objects in the file and to use Rivet's Python API to extract the weight names. The result is then turned into a Python `set` in order to remove duplicate entries in the array. You will notice that most of the scale variations seem to follow the pattern `MUR*_MUF*_PDF261000` at their core where the wild card is either `0.5`, `1` or `2`. We can use this observation to construct a regular expression that will select all the relevant weights

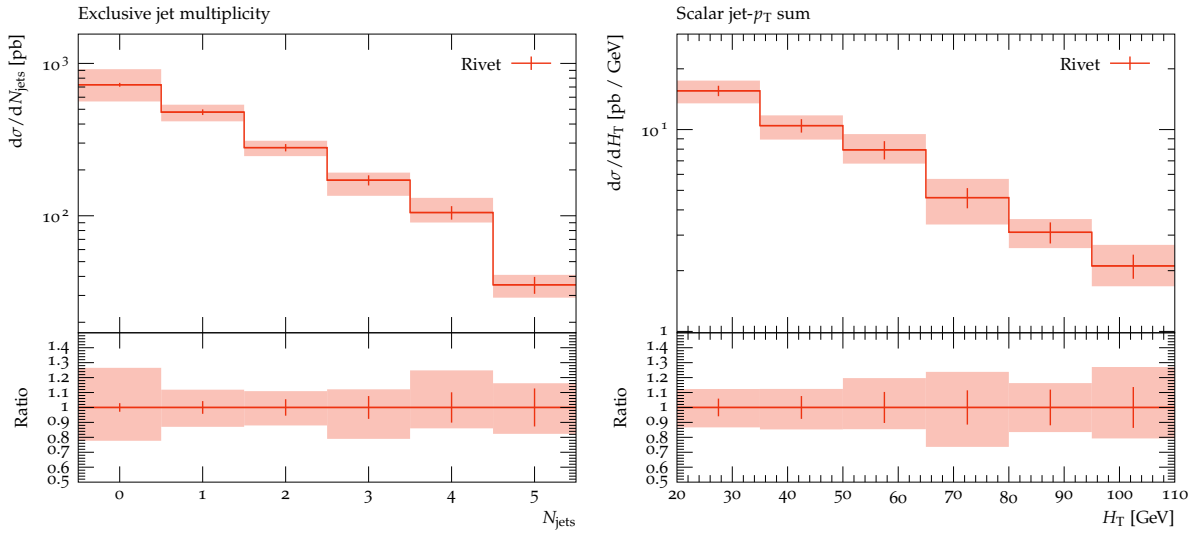
```
rivet-mkhtml --with-variations Rivet.yoda:"Variations=.*MUR.*MUF.*PDF261000.*"
```

This will essentially produce the same plots as above but without the outliers at the bottom of the canvas. Let's take it one step further and attempt to combine the weights into a band. In general, the prescription for how to combine the scale and PDF variations depends on the specific setup, in particular the PDF set used and the precise format of the weight names. The combination strategy needs to be defined by the user in the end, but Rivet supports the most common prescriptions out of the box. The simplest thing one could ask is to combine the weights into an uncertainty band based on the envelope of the scale variations. For instance, we could ask Rivet to work out the envelope for the same regular expression as we had above, like so

```
rivet-mkhtml Rivet.yoda":BandComponentEnv=MUR.*MUF.*PDF261000"
```

which will produce something like the following:

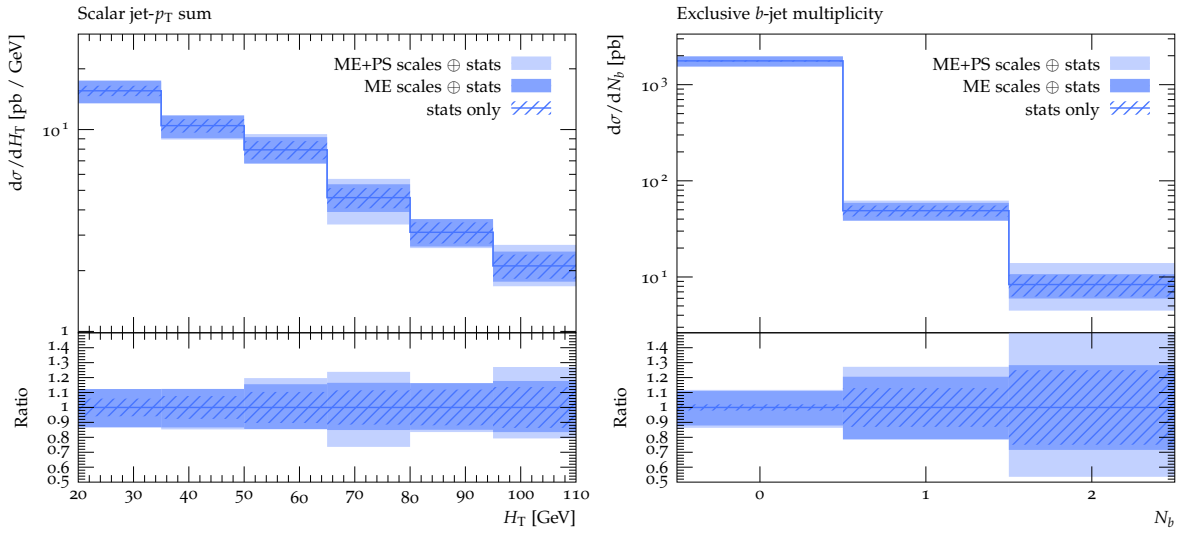
[3] See arXiv:2203.08230.



This is not too bad for a first quick look at the rough size of the uncertainties, but we can do better with a little bit more work. In particular, if we take another close look at the list of weight names available for this setup, we see versions with and without an additional `ME_ONLY_` prepended to the name. The difference between these is that the scales were only varied in the matrix element or coherently in both matrix element and parton shower. It might be interesting to separate these out into two bands[4].

Exercise 6
Complete the code in the `plotBands.sh` skeleton.

Once the weights are combined into bands, it is straightforward to superimpose different bands using the usual plotting scripts. In the example above, we also play around with different colours, band styles and opacity levels to give an idea for how to do a few more fancy style customisations with the plotting scripts in general. The result should look something like this



Even with this small event sample, it can be seen that the parton-shower-based scale variations become relevant in the more exclusive regions of phase space.

[4]There are different ways to go about this. In this tutorial, we try to keep it simple and do everything via the command line using a single output file. For more complicated prescriptions, it might be more convenient to play around with the Python API and use it to combine the weights in more complex ways and/or to write out a separate output file for different uncertainty components in order to keep things potentially more organized.

At this stage you should be familiar with the basics of running Rivet, be able to write a custom routine, make some first plots and even construct uncertainty bands from the weight variations. If you have some spare time, feel free to have a go at the next section which will take you through the basics of homogeneous and heterogeneous merging.

9 Merging parallel runs and different processes

In practice, we rarely have to deal with a single event file, over which we can run our analysis. Typically different processes are generated separately, giving rise to multiple events files with different process-dependent cross-sections associated with each of these files. We can then run our analysis over each of these files and thereby estimate the contribution of the respective process to the region of phase space that we are interested in. By stacking all contributing processes on top of each other – weighted by cross-section – we would arrive at the total prediction. The stacking of different processes is an additive combination that is sometimes referred to as *heterogeneous* or *non-equivalent* merging.

What is more, we might need so many events to be able to arrive at a smooth prediction that storing the events in one huge event file is just not practical and we find it more efficient to produce a given process in many parallel runs, leading to many output event files for the process under consideration. This has the advantage that we can also run our analysis in a parallelised fashion, thereby reducing the overall processing time greatly. The downside is that we end up with many output histogram files when really we only want a single one. Combining output histogram files in a statistically correct way as though they had come from a single run is sometimes referred to as *homogeneous* or *equivalent* merging. Let's take a look at both cases.

9.1 Equivalent merging

We can emulate a parallel run by splitting our run in half as follows

```
rivet --pwd -n 5000 -o part1.yoda.gz -a MY_ANALYSIS Zjets13TeV_10k.hepmc.gz
rivet --pwd --nskip 5000 -o part2.yoda.gz -a MY_ANALYSIS Zjets13TeV_10k.hepmc.gz
```

where the first command only runs over the first 5000 events and the second skips the first 5000 events. In a sense, this is very similar to running the analysis multiple times on a subset of the files from a big pool of events files corresponding to the same process. In order to merge them correctly, one should be mindful of the `finalize` part of the a given routine: That is the part of the analysis where histgrams are normally scaled to cross-section or normalised to unity, or where ratios or efficiencies are constructed. In general, these are post-processing operations that should really be applied to the *merged* histograms, and so the art of equivalent merging is to first undo whatever scaling has been applied in the `finalize` method, then to stack the pre-`finalize`d histograms before finally reapplying the `finalize` logic to the stacked objects. For this reason, Rivet will write out every booked histogram twice, correspponding to the state of the object before and after running `finalize` respectively. The pre-`finalize` version will come with the prefix `/RAW` in the path – check it out for yourself![5] In order to perform a statistically correct equivalent merging, we want to stack the raw histograms and simply rerun the `finalize` part of the routine over the stacked objects. The following command will achieve this:

```
rivet-merge -e -o merged.yoda.gz part1.yoda.gz part2.yoda.gz
```

where the `-e` (or `--equiv`) option denotes equivalent merging.[6] We can easily verify that this gives the same result as our original output file:

```
rivet-mkhtml Rivet.yoda:"Title=single run" merged.yoda.gz:"Title=merged run"
```

which should give you a straight horizontal line at unity in the ratio panel.

[5] This trick also enables us to interrupt a Rivet run at any point and pick it up where we left off – also known as *reentrant histogramming*.

[6] You might notice a warning message about reentrant safety, which Rivet will print when it doesn't recognise a routine, e.g. because it cannot find the associated `.info` file. Reentrant histogramming can only work if all the information need to pick up the run where it was interrupted is provided in the output file. If the logic applied in `finalize` depends on information which can only be derivef from the event file, then the reentrant histogramming will not be possible.

9.2 Non-equivalent merging

We can use our lepton-mode option to produce separate output histogram files for the electron and the muon channel as follows:

```
rivet --pwd -o zee.yoda.gz -a MY_ANALYSIS:LMODE=EL Zjets13TeV_10k.hepmc.gz
rivet --pwd -o zmm.yoda.gz -a MY_ANALYSIS:LMODE=MU Zjets13TeV_10k.hepmc.gz
```

This is almost as though we had been given separate HepMC event files for each lepton channel, except then we wouldn't have had to specify an option since the event sample would only contain one lepton flavour to begin with.

In order to stack the histograms in the two output files, simply run

```
rivet-merge --merge-option LMODE -o zll.yoda.gz zee.yoda.gz zmm.yoda.gz
```

where the `-e` flag from the previous section is now missing since non-equivalent merging is the default mode of `rivet-merge`. Instead we use the optional `--merge-option` flag here to remove the option string from the histogram path. We can again easily verify that this gives the same answer as our original output file:

```
rivet-mkhtml Rivet.yoda:"Title=both channels" zll.yoda.gz:"Title=stacked channels"
```

where `Rivet.yoda` is the output from running the routine without options over the HepMC event file.

10 Summary

In this tutorial, we covered various practical aspects of a typical truth-level analysis as well as common issues to do with fiducial particle-level definitions. A Rivet routine is a snippet of C++ code that exactly implements the analysis logic. We wrote a simple routine to analyse events of weak-boson production in association with jets using Rivet. (An example solution will be made available after the last tutorial session.) Along with making the measurement results publicly available on HEPData, providing a Rivet routine is an important aspect of analysis preservation that helps to maximise the impact of the analysis results as well as making them overall more useful to the community.