version 1: EICUG 2020

A brief introduction to RIVET for the EIC

Christian Bierlich, Lund University
Email: bierlich@thep.lu.se

Contents

1 Summary

This tutorial will introduce you to the basic elements of RIVET, with emphasis on DIS functionality. After a short introduction to the paradigm, it will include several exercises, that assumes that you have RIVET [1] (version 3.1 or higher), as well as a Monte Carlo event generator capable of generating HEPMC events available. Follow the instructions in section 2 if you do not already have this. You will learn:

- To run an analysis and plot the results.
- To modify your resulting figures.

- To write your own analysis to display your observables of choice.
- To use more advanced analysis techniques, such as percentile binning, analysis options and jet finding with FastJet [2].

While this document is meant to be self-contained enough to be useful for self-study, it is by no means a replacement of either the online tutorial documentation available at `https://gitlab.com/hepcedar/rivet/tree/master/doc/tutorials`, nor the documentation included in the Rivet manual [1] or the specific manual for heavy ion features [3]. (Only in very rare cases is it recommended to follow manuals for versions older than v. 3, as they are superseded by newer ones.) To follow this tutorial in a purely self-contained way, one can go to the tutorial url above, and follow the steps necessary for installation outlined there, before proceeding to the next section.

2 Setting up Rivet

Since Rivet relies on several third party libraries, it can be cumbersome to install. Several solutions exist for overcoming this problem. For tutorial purposes, the easiest thing is to rely on a Docker image. If you have participated in the eic Pythia8 tutorial, you have a similar setup as presented on the Rivet tutorials page. If you have not, go to the page, and follow the instructions.

If you followed the Pythia8 tutorial, go to the same working directory in a terminal (in the following instructions, terminal commands are prefaced with a `$>`), and create aliases for running Rivet from *inside* the container on *your own* machine (where `<image name>` is the name of the Docker image you pulled):

```
$> alias rivet='docker run -i  --rm  -u `id -u $USER`:`id -g` \
    -v $PWD:$PWD -w $PWD <image name> rivet'
$> alias rivet-mkanalysis='docker run -i  --rm  -u `id -u $USER`:`id -g` \
    -v $PWD:$PWD -w $PWD <image name> rivet-mkanalysis'
$> alias rivet-build='docker run -i  --rm  -u `id -u $USER`:`id -g` \
    -v $PWD:$PWD -w $PWD <image name> rivet-build'
$> alias rivet-mkhtml='docker run -i  --rm  -u `id -u $USER`:`id -g` \
    -v $PWD:$PWD -w $PWD <image name> rivet-mkhtml'
```

3 A brief introduction to the Rivet paradigm

Rivet is an analysis framework intended for analyzing Monte Carlo events, producing a range of observables, putting them into histograms and comparing them to data. As such, it is at face value not much different than many other analysis frameworks, such as *e.g.* Root, or simply generating Monte Carlo events, and filling them directly into histograms.

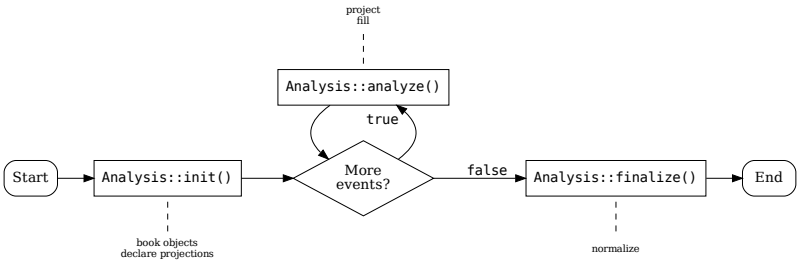

Figure 1: Sketch of the RIVET program flow, with initialization, analysis and finalization steps, as described in the text. Figure from [3].

What sets RIVET apart, is rather its imposed constraints, which forces the user to only calculate experimentally measurable quantities. As such, RIVET does best either as a tool to compare a prediction from a Monte Carlo event generator to data, or as a tool to provide predictions which are guaranteed to be well-defined experimentally.

This is to a high degree achieved by the so-called `Projection` abstraction, which is central to RIVET. Each `Projection` calculates a (set of) physical observables, *e.g.* the set of all final-state charged particles meeting some kinematical cuts, the set of all possible reconstructed Z-bosons, an event shape or a collection of jets. From the `Projection`, desired quantities can then be extracted and filled into histograms. Besides providing an abstraction which demands calculation of only experimentally observable quantities, the `Projection` paradigm also offers several computational advantages, as described in ref. [1].

Technically, the framework is written as a C++ shared library, with most user-facing calls implemented in Python wrappers. Analyses are written as C++ plugin libraries, usually with one implemented RIVET analysis corresponding to one published paper. There are currently over 900 such analyses already implemented in the framework, from a large variety of experiments and beam configurations. A RIVET analysis is implemented as a C++ class, inhering from an `Analysis` base class, and split into three parts, as shown in figure 1:

Initialization: The `init()` method is called once per analysis, and is used to declare ("book") histograms and projections.

Analysis: The `analyze(const Event&)` method is called once per event. Here single event quantities are calculated from the booked projections, and filled into histograms.

Finalization: The `finalize()` method is called once, at the end of each analysis, and is intended to apply normalization to histograms, fill histograms with all-event averages, or construct ratios.

Once the analysis is done, the results are written to a file in the `.yoda` file format. This is a human readable format for histograms, which can be plotted directly with provided scripts.

4 Running your first DIS analysis

In this section we will run two analyses suitable for electron-proton collisions. It is preferable to use an event sample generated by yourself. If you don't have such an event sample, a PYTHIA8 event sample (50k events, 168 MB) can be obtained with:

```
$> wget http://home.thep.lu.se/~bierlich/eic/pythia-dis.hepmc.gz
```

Meanwhile, the basic functionality can be tested. Call the RIVET `help` function, as well as taking a look at available analyses:

```
$> rivet --help
$> rivet --list-analyses
```

Run two analyses over the events. The first, called `MC_DIS_Check`, is a test analysis which we will return to, the other, called `H1_1994_S2919893`, is an energy flow measurement from H1 [4]. Once they are done, plot and view the results in a web browser. (Substitute the name of the event file with the name of your event file, if you have one).

```
$> rivet pythia-dis.hepmc.gz -a MC_DIS_Check -a H1_1994_S2919893
$> rivet-mkhtml Rivet.yoda
$> firefox rivet-plots/index.html
```

Take a look at the figures generated by the two analyses. In `MC_DIS_Check`, you see distributions of the basic DIS kinematics variables, x, y and Q^2. A cut on Q^2 imposed on the generator level, should be clearly visible. Importantly, the values shown here are not the true values which entered the calculation, but rather reconstructed values from final state particles. Even though most DIS events will only have a single obvious candidate for the scattered lepton, the event can include more. An analysis wanting to apply the same observable definitions to theory and data, must therefore implement a way to distinguish between these, in order to resolve any ambiguity. All this is done automatically in RIVET, with options to select the highest energy lepton (default), or by rapidity or transverse energy. This is taken care of by the projection `DISKinematics`. Take this opportunity to use the online RIVET documentation to study the analysis code, as well as the header of the `DISKinematics` projection.

Analysis code: Go to `https://rivet.hepforge.org/`, and click on *Analyses* → *Standard analyses* in the left menu, which should take you to this page. Find the analysis name, click it, and study the code. Note in particular:

1. Declaration of the two projections `DISLepton` and `DISKinematics` in `init()`, and the following application of `DISKinematics` in `analyze(const Event&)`. The latter step is necessary if the analysis step should be allowed to access members of `DISKinematics`.
2. Booking of histograms in `init()`. Pointers to histograms (`Histo1DPtr`s) are declared as members of the class, and booked with a name and axis definitions.
3. Filling of histograms in `analyze(const Event&)`. The `fill(double)` method of the histogram is called, with the appropriate value as an argument. (Note: Some Monte Carlo event generators deliver events with event weights. These are handled automatically by RIVET, and the user should not worry about it when filling histograms).

Projection header: Click on *Doxygen code/API docs* in the left menu, and then on *Class index*. Clicking on the projection name, should take you to this page. Here you can take a look at the methods defined in this class, which includes both the normal kinematical quantities, as well as methods to return Lorentz transformations from the lab frame (default), to the hadronic center of mass frame and the Breit frame. These will be used in section 6.

In `H1_1994_S2919893`, a full analysis, including comparison to data taken by H1, has been performed. In the plots you already see data points, which are loaded automatically by the analysis. Following the same steps as above, take a look at the analysis code. Note several differences from the previous analysis (apart from the fact that this real analysis is much more complicated):

1. Histogram booking is different. Instead of supplying values for histogram axis, three integers are provided. This is the naming scheme for the data reference, which is in turn taken from https://www.hepdata.net/. By providing these indices in the booking, axis are defined in the same way as the reference, and comparison will be done automatically.

2. Another projection, the `FinalState`, is used. It contains all particles of the final state, which are here analysed. Find its use in the `analyze(const Event&)` method.

3. The method extracting the Lorentz transform to the hadronic center of mass system is used to boost the momentum of final state particles. Find its use.

4. In `finalize()` histograms are no longer just normalized, but normalized to the average number of charged particles per event.

You have now been through a simple analysis, as well as a realistic one, which also compares to data. It is now time to write some code yourself.

5 Analysis and plot modifications

The goal of this section is to construct a modified version of `MC_DIS_Check`, and run it. We use here the name `MC_DIS_Mod`, but you can of course substitute any name you like. Start by making an analysis template, by:

```
$> rivet-mkanalysis MC_DIS_Mod
```

This creates three files in you working directory. `MC_DIS_Mod.cc`, which will contain the source code of the analysis, `MC_DIS_Mod.info`, which will contain some general information about the analysis, and `MC_DIS_Mod.plot`, which will contain plotting information for the analysis figures.

Open the `.cc` file with your preferred text editor. It contains a show-case analysis, which is of less importance here. Substitute the contents of the three member methods, the `#include` statements, as well as the histogram pointers declared at the bottom, with the contents of `MC_DIS_Check`, which you looked at before.

Add another histogram to the analysis, and fill it with values for W^2, as computed by the `DISKinematics` projection – re-check the header in the documentation if needed. To analyze the Monte Carlo events with your modified analysis, you must first go through a few steps. The analysis must be compiled, and Rivet must be made aware that an analysis plugin exists in this directory, by using the `--pwd` option on calls to Rivet[1].

```
$> rivet-build RivetMC_DIS_Mod.so MC_DIS_Mod.cc # Compile the analysis
$> rivet --list-analyses --pwd | grep MC_DIS_Mod # Check analysis is found
$> rivet pythia-dis.hepmc.gz -a MC_DIS_mod --pwd # Run the new analysis
```

While the analysis runs, open the file `MC_DIS_Mod.plot`. This encodes plotting information for your modified analysis. To, for example, plot the x_{Bj} histogram as in the test analysis (with a logarithmic horizontal axis), you would put:

```
BEGIN PLOT /MC_DIS_Mod/xBj
LogX=1
END PLOT
```

Take a look at the documentation for plotting options here, and modify to your liking. There are a large number of options. Once the analysis is finished running, plot it, remembering to tell Rivet where to find your modified `.plot` file:

```
$> rivet-mkhtml Rivet.yoda --pwd
```

[1] If you had Rivet installed on your own system, you could simply tell Rivet to do this once and for all, by doing `$> export RIVET_ANALYSIS_PATH=$PWD`. But alas, that does not work when you run the commands from a Docker container.

6 Writing your own analysis

In this section, you will extend your modified analysis to a more realistic analysis of final state particles. At this point there is no need to make a whole new analysis file, you can simply add some more figures to the existing one. For the purpose of this tutorial, focus will rather be on showcasing functionality than providing the observable most interesting to your own interest – but feel free to add your own!

We will add two histograms containing the charged hadron multiplicity per event as a function of pseudo-rapidity (η) in the hadronic center of mass systems. One for $x < 10^{-3}$, and one for $x \geq 10^{-3}$. Start by declaring a `DISFinalState` projection in `init()`, to get access to final state particles. This projection will automatically remove the DIS lepton, and has built-in functionality to deliver final state particles in the desired boost frame. You can look up the projection in the online documentation as before.

```
// Add to includes
#include "Rivet/Projections/DISFinalState.hh"
...
// Add in init
declare(DISFinalState(DISFinalState::BoostFrame:HCM), "DISFS");
```

Next you want to book the two histograms, as well as two `CounterPtr`s, the latter keeping track of the number of events filled into each histogram[2]. Book both in `init()`, and remember to declare them in the class, preferably in the end along with rest of the histogram pointer declarations.

```
// Book in init
book(_hist_multeta1, "multeta1", 30, -6, 6);
book(_hist_multeta2, "multeta2", 30, -6, 6);
book(sow1, "_sow1");
book(sow2, "_sow2");
...
// Remember to declare as class members.
Histo1DPtr _hist_multeta1, _hist_multeta2;
CounterPtr sow1, sow2;
```

By prefacing the counter names with an underscore, we prevent them from being plotted with `rivet-mkhtml`. (After all, they are not the main purpose here.)

The projection should now be applied, and the histograms filled, in the analysis step. Apply the projection below the application of `DISKinematics`, by inserting the following line:

[2]The preference is towards using counters rather than raw `double`s, since the former are streamed to the output file, which thus can be properly merged with other samples in a parallel run. Using counters, you also don't have to worry about event weights.

```
const DISFinalState& disfs = apply<DISFinalState>(event, "DISFS");
```

Fill the counters and histograms by adding the following lines below the filling of previous histograms.

```
if (x < 1e-3) sow1->fill();
else sow2->fill();
for (auto p : disfs.particles()) {
  if (p.isHadron() && p.isCharged()) {
    if (x < 1e-3) _hist_multeta1->fill(p.eta());
    else _hist_multeta2->fill(p.eta());
  }
}
```

Finally, the histograms must be scaled by the contents of the counters in `finalize()`. Add the following two lines:

```
scale(_hist_multeta1, 1./sow1->sumW());
scale(_hist_multeta2, 1./sow2->sumW());
```

You should now be ready to re-compile the analysis, run it and plot the results. Modify the `.plot` file to your liking, and if you want to have a description of the analysis displayed on the plot page, the `.info` file should be modified accordingly.

```
$> rivet-build RivetMC_DIS_Mod.so MC_DIS_Mod.cc
$> rivet pythia-dis.hepmc.gz -a MC_DIS_Mod --pwd
$> rivet-mkhtml Rivet.yoda --pwd
```

You are now done with this part of the task. If you have time, you can proceed to the next section and add more advanced features. For your reference, the full code of the analysis (which will compile and run) at this point, is pasted in the appendix A.

7 More advanced analysis techniques

In this section, three more advanced analysis techniques are introduced. The pace is stepped up a bit, and it is implied that one should study the online code documentation if something is unclear. You can pick none, one, two or all of the techniques and implement them, according to taste.

7.1 Analysis options

It is possible to specify options to an analysis, thereby allowing the user to interact with the analysis while calling it from the command line. This can be useful to change parameters

to algorithms, cuts etc., without having to rewrite or recompile an analysis. This will be exemplified in the following, by adding an option to boost the `DISFinalState` to the Breit frame.

Begin by adding the option to `init`. An option can be any data type which can be streamed from the command line, even user defined ones, but here we will keep it to strings, by adding the following right before declaring `DISFinalState` in `init`:

```
string boostSystem = getOption<string>("boost","hcm");
```

This allows the analysis to get an string data type option named `boost` from the command line, holding the default value `hcm`. Let the declaration of `DISFinalState` depend on this option, by replacing the current declaration with:

```
if (boostSystem == "breit")
  declare(DISFinalState(DISFinalState::BoostFrame::BREIT),"DISFS");
else
  declare(DISFinalState(DISFinalState::BoostFrame::HCM),"DISFS");
```

Finally, the analysis `.info` file must be modified to accept this option. Open the file, and add a new section just before `Description:`, saying:

```
Options:
 - boost=hcm,breit
```

Recompile the analysis. The options can now be specified on the command line, and it is even possible to run the analysis twice – once with each option – by specifying the analysis twice.

```
$> rivet pythia-dis.hepmc.gz -a MC_DIS_Mod:boost=hcm \
    -a MC_DIS_Mod:boost=breit --pwd
$> rivet-mkhtml Rivet.yoda --pwd
```

You will see that the results are now plotted on top of each other for the two option selections.

7.2 Percentile binning

In heavy ion physics, it is common to bin an observable in percentiles of another observable, *e.g.* as collision "centrality". This is possible to do in Rivet as well, by constructing a calibration analysis used to calculate this observable using Monte Carlo, and calibrate against it afterwards[3]. We will here just pick a rather silly example, and construct the

[3]It is also possible to calibrate against experimentally defined curves, impact parameter, or even supply a calibration value directly. Study the full manual for such use cases.

charged hadron multiplicity as function of η (for all x, in the hadronic center of mass system) in percentile bins of 0-20%, 20-40%, 40-60% and 60-100% of the charged hadron multiplicity at $|\eta| > 3$.

In order to bin in percentiles of some quantity, first a `SingleValueProjection` projecting this quantity must be defined. Secondly, a calibration analysis must be defined. For the purpose of this tutorial, they can all be put in the same `.cc` file as the analysis, but normally they would be separated out, as one experiment needs only be concerned with a single implementation of both for most analyses. Add

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

to the include statements, and insert the code for the projection and calibration analysis before your own analysis. Read through it, but don't be concerned with the details. As stated, implementing projections and calibration analyses are rarely needed – when they exist once, they can be reused in the body of one's own analysis:

```
class MC_DIS_CALIB : public SingleValueProjection {
public:
  MC_DIS_CALIB() {
    declare(DISFinalState(DISFinalState::BoostFrame::HCM),"DISFS");
  }
  virtual ~MC_DIS_CALIB() {}

  DEFAULT_RIVET_PROJ_CLONE(MC_DIS_CALIB);

protected:
  void project(const Event& e) {
    clear();
    int nch = 0;
    for (auto p : apply<DISFinalState>(e, "DISFS").particles())
      if (p.isHadron() && p.isCharged() && abs(p.eta()) > 3.) ++nch;
    set(nch);
  }
  virtual CmpState compare(const Projection& p) const {
    return mkNamedPCmp(p, "DISFS");
  }
};

class MC_DIS_PERC : public Analysis {
public:
  DEFAULT_RIVET_ANALYSIS_CTOR(MC_DIS_PERC);
```

```
    void init() {
      declare(MC_DIS_CALIB(),"PERC");
      book(_fm, "FM",25,0,25);
    }
    void analyze(const Event& event) {
      _fm->fill(apply<MC_DIS_CALIB>(event, "PERC")());
    }
    void finalize() {
      _fm->normalize();
    }
    Histo1DPtr _fm;
  };
  DECLARE_RIVET_PLUGIN(MC_DIS_PERC);
```

We will order the percentile binned histograms and counters in C++ `std::map`s, and define a vector holding the upper limits of the intervals. Add the following members to your analysis class:

```
map<double, Histo1DPtr> histEtaPerc;
map<double, CounterPtr> sowPerc;
vector<double> percentileBins;
```

Declare a `CentralityProjection` in `init()`, and book histograms and counters:

```
declareCentrality(MC_DIS_CALIB(), "MC_DIS_PERC", "FM", "PERC");
percentileBins = {20, 40, 60, 100};
for (size_t i = 0; i < percentileBins.size(); ++i) {
  double pb = percentileBins[i];
  book(histEtaPerc[pb], "histEtaCent-"+toString(pb), 30, -6, 6);
  book(sowPerc[pb], "_sow"+toString(pb));
}
```

In the analysis step, find the histogram and counter corresponding to the current percentile bin, by calling the `CentralityProjection`:

```
const CentralityProjection& cent =
  apply<CentralityProjection>(event, "PERC");
double c = cent();
auto hItr = histEtaPerc.upper_bound(c);
if (hItr == histEtaPerc.end()) return;
auto sItr = sowPerc.upper_bound(c);
if (sItr == sowPerc.end()) return;
```

Fill the counter outside the particle loop, and the histogram inside, by calling the fill methods `sItr->second->fill();` and `hItr->second->fill(p.eta());` respectively. Finally normalize the histograms by their appropriate counter, by placing the following loop in `finalize()`

```
for (size_t i = 0; i < percentileBins.size(); ++i) {
  double pb = percentileBins[i];
  histEtaPerc[pb]->scaleW(1./sowPerc[pb]->sumW());
}
```

Before the analysis can be run, the `.info` file must be appended. Open it in a text editor, and add a new section just before `Description:`:

```
Options:
 - cent=GEN
```

If you have already done the part about analysis options, you can just append the last line to your existing `Options` section.

You can now recompile and run the analysis. First the calibration analysis must be run, and the resulting `.yoda` file from that, must be pre-loaded into the second run. We give the calibration file the name `calib.yoda`, by specifying an output name:

```
$> rivet-build RivetMC_DIS_Mod.so MC_DIS_Mod.cc
$> rivet pythia-dis.hepmc.gz -a MC_DIS_PERC -o calib.yoda --pwd
$> rivet pythia-dis.hepmc.gz -a MC_DIS_Mod:cent=GEN -p calib.yoda --pwd
$> rivet-mkhtml Rivet.yoda calib.yoda --pwd
```

And the resulting figures can be studied.

7.3 Jet finding with FastJet

Rivet comes with an interface to the FastJet library for finding jets. The library, and the user supplied contributed algorithms in `fjcontrib` provides a wealth of possibilities – far too much to cover here. For other use cases than the simple one covered here, you are thus referred to the extensive documentation available online, or from existing Rivet analyses implementing one.

We will add a jet finder to the analysis, with the goal of finding $k_\perp$-jets with an R-parameter of 1 and $E_\perp > 4$ GeV. We will plot the distribution of number of such jets, as well as the $E_\perp$ distribution.

First the header for the FastJet interface must be included, add:

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

For the `FastJets` projection to be declared, it must have access to an underlying projection in `init()`, here the `DISFinalState`. Therefore, remove the line where the `DISFinalState` projection is declared, and replace it with:

```
const DISFinalState& disfs =
  declare(DISFinalState(DISFinalState::BoostFrame::HCM),"DISFS");
declare(FastJets(disfs, fastjet::JetAlgorithm::kt_algorithm,
  fastjet::RecombinationScheme::pt_scheme, 1.0,
  JetAlg::Muons::ALL, JetAlg::Invisibles::NONE, nullptr), "JETS");
```

The second line declares the `FastJets` projection. If any of the arguments to the constructor prompt doubts, they can be looked up in the online documentation. Also, book two histograms to fill in the analysis step (one for the $E_\perp$ distribution, one for number of jets):

```
// In init().
book(_hist_njets, "njets", 5, 0, 5);
book(_hist_jetET, "jetET", 20, 4, 60);
...
// Declarations as class members.
Histo1DPtr _hist_njets, _hist_jetET;
```

In the analysis step, the jets can now be accessed like any other projection. The application of the projection, and histogram filling, should look like this:

```
Jets jets =
  apply<FastJets>(event, "JETS").jets(Cuts::Et > 4. * GeV, cmpMomByEt);
_hist_njets->fill(jets.size());
if (jets.size() == 0) return;
_hist_jetET->fill(jets[0].Et());
```

And in `finalize()` the histograms for number of jets should be normalized to unity, and the $E_\perp$ histogram to the cross section (here in picobarn):

```
normalize(_hist_njets);
scale(_hist_jetET, crossSection()/picobarn/sumOfWeights());
```

The analysis can now be compiled and run like normally. (Note, that if you have already done the percentile calibrated observables, you have to specify the preload file and the `cent=GEN` option as before).

```
$> rivet-build RivetMC_DIS_Mod.so MC_DIS_Mod.cc
$> rivet pythia-dis.hepmc.gz -a MC_DIS_Mod --pwd
$> rivet-mkhtml Rivet.yoda --pwd
```

References

[1] C. Bierlich *et al.*, "Robust Independent Validation of Experiment and Theory: Rivet version 3," *SciPost Phys.*, vol. 8, p. 026, 2020.

[2] M. Cacciari, G. P. Salam, and G. Soyez, "FastJet User Manual," *Eur. Phys. J.*, vol. C72, p. 1896, 2012.

[3] C. Bierlich *et al.*, "Confronting experimental data with heavy-ion models: RIVET for heavy ions," *Eur. Phys. J.*, vol. C80, no. 5, p. 485, 2020.

[4] I. Abt *et al.*, "Energy flow and charged particle spectrum in deep inelastic scattering at HERA," *Z. Phys.*, vol. C63, pp. 377–390, 1994.

A Full source code for section 6

A complete RIVET analysis answering section 6 is given below, for debugging purposes.

```
#include "Rivet/Analysis.hh"
#include "Rivet/Projections/DISKinematics.hh"
#include "Rivet/Projections/DISFinalState.hh"

namespace Rivet {

  class MC_DIS_Mod : public Analysis {
  public:

    /// Constructor
    DEFAULT_RIVET_ANALYSIS_CTOR(MC_DIS_Mod);

    /// Initialize histograms, projections etc.
    void init() {
      DISLepton lepton(options());
      declare(lepton, "Lepton");
      declare(DISKinematics(lepton), "Kinematics");
      declare(DISFinalState(DISFinalState::BoostFrame::HCM),"DISFS");

      // Book histograms
      book(_hist_Q2, "Q2",logspace(100,0.1, 1000.0));
      book(_hist_y, "y",100,0.,1.);
      book(_hist_x, "xBj",logspace(100,0.00001, 1.0));
      book(_hist_W2, "W2", logspace(100, 0.1, 10000.));
```

```
  book(_hist_multeta1, "multeta1", 30, -6, 6);
  book(_hist_multeta2, "multeta2", 30, -6, 6);
  book(sow1, "_sow1");
  book(sow2, "_sow2");
}

/// Perform the per-event analysis
void analyze(const Event& event) {
  const DISKinematics& dk = apply<DISKinematics>(event, "Kinematics");
  const DISFinalState& disfs = apply<DISFinalState>(event, "DISFS");
  if ( dk.failed() ) return;
  double x  = dk.x();
  double y  = dk.y();
  double Q2 = dk.Q2();
  double W2 = dk.W2();
  _hist_Q2->fill(Q2);
  _hist_y->fill(y);
  _hist_x->fill(x);
  _hist_W2->fill(W2);
  if (x < 1e-3) sow1->fill();
  else sow2->fill();
  for (auto p : disfs.particles()) {
    if (p.isHadron() && p.isCharged()) {
      if (x < 1e-3) _hist_multeta1->fill(p.eta());
      else _hist_multeta2->fill(p.eta());
    }
  }
}

/// Normalise histograms etc., after the run
void finalize() {
  normalize(_hist_Q2); // normalize to unity
  normalize(_hist_y); // normalize to unity
  scale(_hist_multeta1, 1./sow1->sumW());
  scale(_hist_multeta2, 1./sow2->sumW());
}

/// The histograms.
Histo1DPtr _hist_Q2, _hist_y, _hist_x, _hist_W2;
Histo1DPtr _hist_multeta1, _hist_multeta2;
CounterPtr sow1, sow2;
```

```
  };

  DECLARE_RIVET_PLUGIN(MC_DIS_Mod);
}
```