Rivet tutorial

Andy Buckley

ATLAS Rivet tutorial, 2011-03-21

Contents

Introduction

What is Rivet?

Rivet is a generator-agnostic validation system for MC generators.

More simply, it's a tool to produce physics plots from an MC generator code which can produce HepMC events. (i.e. every ATLAS generator – see `Generators/Rivet_i`)

This is useful for validating generators – only need to write the analysis once and it can be used to validate and compare every generator that should be able to simulate it.

Policy in ATLAS Standard Model group is that each data analysis should produce a Rivet MC analysis for analysis and MC validation.

Also useful as an input to MC tuning.

Some more on Rivet's design

MC analysis system operating on HepMC events. Intentionally ignorant of what generator produced the events it sees.

Emphasis on not messing with the MC implementation details: actually reconstruct bosons, don't trace back partons, etc. Life is (eventually) simpler this way!

Lots of standard analyses built in, including key ones for pQCD and MPI model testing. New analyses can be picked up at runtime: nice API with lots of tools to make this as simple and pleasant as we can. Computations automatically cached. Histograms automatically synchronised.

Please write Rivet analyses of your analysis and contribute them (to the ATLAS MC tuning group in first instance).

Latest version is 1.5.0.

Setup

Rivet docs: online at **http://projects.hepforge.org/rivet/** – PDF manual, HTML list of existing analyses, and Doxygen.

Log in to lxplus. If you're not running bash shell (`echo $SHELL`), then run `bash` to make life more pleasant.

- Set up to use the ATLAS GCC and Python versions (not the SLC5 defaults), e.g.

  `asetup 16.6.3` (alternatively, source the LCG GCC 4.3 and Python 2.6 setup scripts by hand)
- Genser Rivet setup:

  ```
  source /afs/cern.ch/sw/lcg/external/MCGenerators/
   rivet/1.5.0/i686-slc5-gcc43-opt/rivetenv.sh
  ```
- Genser AGILe setup:

  ```
  source /afs/.cern.ch/sw/lcg/external/MCGenerators/
   agile/1.2.0/i686-slc5-gcc43-opt/agileenv.sh
  ```

Test commands: `rivet --help` `agile-runmc --help`

First Rivet runs

Viewing available analyses

Rivet knows all sorts of details about its analyses!

- List available analyses:
  `rivet --list-analyses`
- List available analyses with a little more detail:
  `rivet --list-analyses -v`
- List ATLAS analyses with a little more detail:
  `rivet --list-analyses -v ATLAS_`
- Show some pure-MC analyses' full details:
  `rivet --show-analysis MC_`

The PDF and HTML documentation is also built from this info, so is always synchronised.

The analysis metadata is provided via the analysis API and usually read from an `.info` file which accompanies the analysis.

Running a simple analysis (standalone)

For simplicity, we get the events from generator to Rivet by writing to a filesystem pipe. NB. This has to live in a non-AFS directory!

```
mkfifo /tmp/$USER/hepmc.fifo
```

We're going to use AGILe to run PYTHIA 6 for demonstration – use the same or run any other generator that you like with HepMC output going to the FIFO:

```
agile-runmc Pythia6:424 --beams=LHC:7000 -n 2000 -o
/tmp/$USER/hepmc.fifo &
```

Now attach Rivet to the other end of the pipe:

```
rivet -a MC_GENERIC /tmp/$USER/hepmc.fifo
```

Tada! You can use multiple analyses at once, change the output file, etc.: see `rivet --help`

Plotting

Sorry, no ROOT! (Well, you can convert the Rivet output with the **`aida2root`** script...)

For now we are using the LWH implementation of the AIDA interfaces. The plots are written out as DataPointSet objects in AIDA XML format. A histogramming upgrade is underway!

Plotting is pretty easy, though:

```
compare-histos Rivet.aida
make-plots *.dat
```

Then view with a file browser/evince/gv/xpdf... **`--eps`**, **`--png`** etc. also work. And **`--help`** is available for all Rivet scripts.

Running a data analysis

We're going to use the ATLAS 900 GeV/7 TeV min bias analysis:

```
rivet --show-analysis ATLAS_2010_S8918562
```

Note that tab completion should work on **`rivet`** options and analysis names.

Now to run it:
agile-runmc command as before, but with **`--beams=LHC:900`**

```
rivet -a ATLAS_2010_S8918562 /tmp/$USER/hepmc.fifo
```

And plot, much as before:

```
compare-histos Rivet.aida
make-plots --pdfpng ATLAS*.dat
```

You could also use **`rivet-mkhtml --pdf Rivet.aida`**

Running Rivet in Athena

Now we'll run some analyses via the Athena Rivet interface, **`Rivet_i`**. You are already set up to use the 16.6.3 release candidate build via **`asetup`**, so now just get the example Rivet job options from the build:

```
get_files -jo jobOptions.rivet.py
```

Modify this JO to run the **`MC_GENERIC`**, **`MC_JETS`**, and **`D0_2004_S5992206`** analyses (the event generator is set to run in Tevatron Run II mode.)

Can you identify and run the ATLAS 2011 version of this dijet $\Delta\phi_{12}$ decorrelation analysis?

Can you find the other example Rivet JO script, to allow reading from evgen POOL files?

Writing a first analysis

Writing an analysis

Writing an analysis is of course more involved than just running **`rivet`**! However, the C++ API is intended to be friendly: most analyses are quite short and simple because the bulk of the computation is in the library.

An example is usually the best instruction: take a look at **`/afs/cern.ch/sw/lcg/external/MCGenerators/rivet/1.5.0/share/src/Analyses/MC_GENERIC.cc`**

Things to note:

- Analyses are classes and inherit from **`Rivet::Analysis`**
- Usual init/execute/finalize-type event loop structure (familiar from Athena)
- Weird *projection* things in **`init`** and **`analyze`**
- *Mostly* normal-looking everything else

Projections – registration

Major idea: **projections**. These are where the computational meat of Rivet resides. They are just observable calculators: given an `Event` object, they *project* out physical observables. They also automatically cache themselves, to avoid recomputation: this leads to the most unintuitive code structures in Rivet.

They are *registered* with a name in the `init` method:

```
void init() {
  ...
  const SomeProjection sp(foo, bar);
  addProjection(sp, "MySP");
  ...
}
```

Projections – applying

Projections were registered with a name. . . they are then applied to the current event, also by name:

```
void analyze(const Event& evt) {
  ...
  const BaseSomeProjection& mysp =
    applyProjection<SomeProjectionBase>(evt, "MySP");
  mysp.foo()
  ...
}
```

We prefer to get a handle to the applied projection as a const reference to avoid unnecessary copying.

It can then be queried about the things it has computed.Projections have different abilities and interfaces: check the Doxygen on the Rivet website.

Final state projections

Rivet is mildly obsessive about only calculating things from final state objects. Accordingly, a *very* important set of projections is those used to extract final state particles: these all inherit from **`FinalState`**.

- The **`FinalState`** projection finds all final state particles in a given η range, with a given p_T cutoff.
- Subclasses **`ChargedFinalState`** and **`NeutralFinalState`** have the predictable effect!
- **`IdentifiedFinalState`** can be used to find particular particle species.
- **`VetoedFinalState`** finds particles *other* than specified.
- **`VisibleFinalState`** excludes invisible particles like neutrinos, LSP, etc.

Most FSPs can take another FSP as a constructor argument and augment it.

Using FSPs to get final state particles

```
void analyze(const Event& evt) {
  ...
  const FinalState& cfs =
    applyProjection<FinalState>(event, "ChgdFS");
  MSG_INFO("Total charged mult. = " << cfs.size());
  foreach (const Particle& p, cfs.particles()) {
    const double eta = p.momentum().eta();
    MSG_DEBUG("Particle eta = " << eta);
  }
  ...
}
```

Note the lovely **`foreach`** macro – from Boost. We are very into the "make simple things simple" philosophy. Please use **`foreach`** when appropriate in any code that you contribute to Rivet.

Physics vectors

Rivet uses its own physics vectors rather than CLHEP. They are a little nicer to use, but basically familiar. As usual, check Doxygen: **http://projects.hepforge.org/rivet/code/dev/**

`Particle` and `Jet` both have a `momentum()` method which returns a `FourMomentum`.

Some `FourMomentum` methods: `eta()`, `pT()`, `phi()`, `rapidity()`, `E()`, `px()` etc., `mass()`. Hopefully intuitive!

Histogramming

AIDA has Histogram1D and Profile1D histograms similar to the core TH1D and TProfile in ROOT.

Histos can be booked via helper methods on the **`Analysis`** base class, which register the histograms at an appropriate path for their parent analysis, e.g. **`bookHistogram1D("thisname", 50, 0, 100)`**. They can also be booked via a vector of bin edges or *autobooked* from a reference histogram.

The histograms have the usual **`fill(value, weight)`** method for use in the **`analyze`** method. There are **`scale()`** and **`normalize()`** methods for use in **`finalize`**.

The fill weight is important! Generators are often run with some kinematic enhancement which has to be offset with a reduced weight. Use **`evt.weight()`**.

Your first analysis

Let's start with a simple "min bias" type of analysis, just plotting some simple particle properties like η, p_T, ϕ, etc. (. . . mean p_T vs. n_{ch} if you're feeling confident!)

To get an analysis template, which you can fill in with an FS projection and a particle loop, run **`rivet-mkanalysis MY_TEST_ANALYSIS`** – this will make the required files.

Before Rivet 1.5.1, you have to edit the generated **`.info`** file before you can run.

Once you've filled it in, you can either compile directly with **`g++`**, using the **`rivet-config`** script as a compile flag helper, or – more helpfully – run

```
rivet-buildplugin RivetMyTest.so MY_TEST_ANALYSIS.cc
```

In this setup, where we're using the 32 bit Rivet on a 64 bit system, add **`-m32`**

To run, first **`export RIVET_ANALYSIS_PATH=$PWD`**, then run **`rivet`** as before.

Writing a data analysis

Starting a data analysis

If you are implementing an analysis for which there is already experimental data in HepData, **`rivet-mkanalysis`** has another nice trick up its sleeve.

Let's reimplement the ATLAS dijet analysis. The SPIRES key for this analysis is 8817804 and it was published in 2010, so we use the standard Rivet naming convention: this analysis will be called **`ATLAS_2010_S8817804`**.

Run **`rivet-mkanalysis ATLAS_2010_S8817804`**. You should now have the **`.cc`** and **`.info`** template files as before, but also a new **`.aida`** file. You can view the contents of this with the **`aida2flat`** or **`aida2root`** script:

```
aida2flat ATLAS_2010_S8817804.aida | less
```

The histogram names in this data file can be used for histogram autobooking.

Histogram autobooking

The final framework feature to introduce is histogram autobooking. This is a means for getting your Rivet histograms binned with the same bin edges as used in the experimental data that you'll be comparing to.

To use autobooking, just call the booking helper function with only the histogram name (check that this matches the name in the reference **`.aida`** file), e.g.

```
_hist1 = bookHistogram1D("d01-x01-y01")
```

The "d", "x" and "y" terms are the indices of the HepData dataset, x-axis, and y-axis for this histogram in this paper.

A neater form of the helper function is available and should be used for histogram names in this format:

```
_hist1 = bookHistogram1D(1, 1, 1)
```

That's it! If you need the bin edges without booking a persistent histogram (e.g. for booking a temporary LWH histogram), use **`binEdges(name)`** or **`binEdges(d,x,y)`**.

Jets (1)

There are many more projections, but one more important set which we'd like to dwell on is those to construct jets. **`JetAlg`** is the main projection interface for doing this, but almost all jets are actually constructed with FastJet, via the explicit **`FastJets`** projection.

The **`FastJets`** constructor defines the input particles (via a **`FinalState`**), as well as the jet algorithm and its parameters:

```
const FinalState fs(-3.2, 3.2);
addProjection(fs, "FS");
FastJets fj(fs, FastJets::ANTIKT, 0.6);
fj.useInvisibles();
addProjection(fj, "Jets");
```

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

Jets (2)

Then get the jets from the jet projection, and loop over them in decreasing p_T order:

```
const Jets jets =
  applyProjection<JetAlg>(evt, "Jets").jetsByPt();
foreach (const Jet& j, jets) {
  foreach (const Particle& p, j.particles()) {
    const double dr =
      deltaR(j.momentum(), p.momentum());
  }
}
```

Check out the `Rivet/Math/MathUtils.hh` header for more handy functions like `deltaR`.

The End