# Rivet tutorial

Andy Buckley

BOOST'12, Valencia, 2012-07-25


# Contents



# Introduction

# The plan

The last one of these took about 4 hours! Not today…

I'm going to demo in parallel setting up and running Rivet on `lxplus` and on my laptop. Follow if you like, but not essential.

- Some background
- Setting up, and querying available analyses
- From PYTHIA to plots
- Writing and running a (substructure) analysis

There is a bit of version mismatch in what I'll show!
For now, do as I say, not as I do!

# What is Rivet?

Rivet is a generator-agnostic validation system for generators.

Co-developed with HepData & HepForge, co-evolved with FastJet, ...

More straightforwardly: it's a tool for making physics plots from generators that can produce events in the HepMC format.

All the "major" generators can do this one way or another: C++ Pythia 8, Sherpa, Herwig++ out of the box, Fortran PYTHIA 6, HERWIG+JIMMY, etc. via AGILe.

Like ROOT, you mean? A bit nicer! Really.
Rivet's been designed (and redesigned) with usability in mind: analysis code should be able to be concise and clear. And it's become a de facto *de facto* standard for LHC analysis archiving.

Used for generator validation, archiving of (LHC) analysis algorithms corresponding to measurement papers. + MC tuning, model development, BSM studies, ...

# 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, avoid touching partons, etc. Life is eventually simpler and better-defined this way.

Lots of standard analyses are built in, including key ones for pQCD and MPI model testing. Now over 200 built-in analyses!

New analyses can be picked up at runtime: there's a nice API with lots of tools to make this as simple and pleasant as we can. Computations automatically cached. Histograms automatically synchronised. Satisfaction automatically guaranteed...

Experimentalists: please write Rivet analyses of your analysis and contribute them $\Rightarrow$ **improving model systematics.**

**Latest version is 1.8.1.**

# Setup

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

Log in to **`lxplus.cern.ch`**. On lxplus, if you're not in a *bash* shell (**`echo $SHELL`**), then run **`bash`** to make life more pleasant: the Rivet toolkit provides contextual command line completion with bash.

- On lxplus: **`source ~abuckley/public/setupRivetProf.sh`**

Test commands:

- **`rivet --help`**
- **`agile-runmc --help`**

If you want to run on your own laptop, please use the Rivet wiki installation instructions. Your life will be much easier if you have CERN AFS mounted so you can pick up the Genser generators.

# 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!
On lxplus: `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 Pythia6:426 --beams=LHC:8000 -n 2000 -o
/tmp/$USER/hepmc.fifo &
```

Now attach Rivet to the other end of the pipe:

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

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

# Feeding LHEF events into Rivet

For fixed-order specialists whose codes output LHEF events rather than HepMC, you can't use Rivet directly. Of course, IR-(very-) unsafe quantities are risky, but you're all consenting adults (right?)

At Les Houches last year I made a mini filter program which will convert LHEF files or streams to HepMC ones:

**http://svn.hepforge.org/rivet/contrib/lhef2hepmc/**

Use it like this:

```
./lhef2hepmc lhef.fifo hepmc.fifo
```

or

```
./lhef2hepmc lhef.fifo - | rivet
```

Maybe some help will be needed with building this program – it's not an official part of Rivet so you have to download and build it by hand. Let us know if you need a hand.

# Plotting

Not ROOT!

Well, you can convert the Rivet output with the `aida2root` script…

Plotting is pretty easy, though:

```
rivet-mkhtml Rivet.aida
```

or, if you want complete control:

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

Then view with a web browser/file browser/evince/gv/xpdf…

A `--help` option is available for all Rivet scripts.

*For now* we are using AIDA histogramming. This has a few benefits (e.g. "Sumw2" is default) and quite a few issues. Run merging can't be done properly with Rivet via AIDA. We're nearly done with a "proper" upgrade – as my laptop demo will accidentally reveal!

# Running a data analysis

We're going to use the ATLAS 7 TeV inclusive jet analysis:

```
rivet --show-analysis ATLAS_2012_I1082936
```

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:7000`***

```
rivet -a ATLAS_2012_I1082936 /tmp/$USER/hepmc.fifo
```

And plot, much as before:

```
rivet-mkhtml -t "ATLAS jets at 7 TeV" Rivet.aida:PY6
```

or

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

# 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/1.8.1/share/src/Analyses/MC_GENERIC.cc` (or the same via http://svn.hepforge.org/rivet/trunk)

Things to note:

- Analyses are classes and inherit from `Rivet::Analysis`
- Usual init/execute/finalize-type event loop structure (certainly familiar from experimental frameworks)
- 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, e.g.

**http://projects.hepforge.org/rivet/code/dev/hierarchy.html**

# 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 $\eta$ 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. Future extension plans involve momentum selector objects, cf. FastJet 3.

# 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()`**.

# A first analysis

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

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

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 MY_TEST_ANALYSIS.cc
```

To run, first `export RIVET_ANALYSIS_PATH=$PWD`, then run `rivet` as before. . . or add the `--pwd` option to the `rivet` command line.

# 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(20*GeV);
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`.

# Jets (3)

For substructure analysis Rivet doesn't provide extra tools: best just to use FastJet directly

```
const PseudoJets psjets = fj.pseudoJets();
const ClusterSequence* cseq = fj.clusterSeq();

Selector sel_3hardest = SelectorNHardest(3);
Filter filter(0.3, sel_3hardest);
foreach (const PseudoJet& pjet, psjets) {
  PseudoJet fjet = filter(pjet);
  ...
}
```

Improvements and suggestions more than welcome!

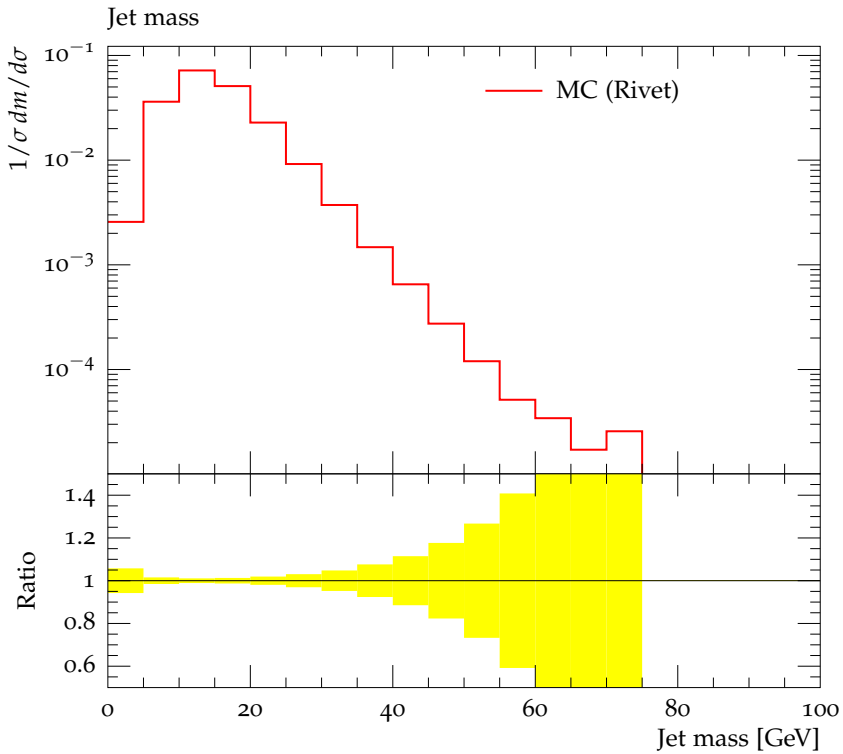
# A jetty analysis

I'll walk through making and running a very simple jet + substructure analysis – but no physics input or complexity here!

The ideal result will be found on AFS at

**`~abuckley/public/rivet-tutorial/`**

Note: if using FastJet3 tools, you'll need to add `lifastjettools` to the `rivet-buildplugin` command line. And a `-L/path/to/` arg as well, until the next release. Just compilation, no magic.



# Writing a *data* analysis

# Starting a data analysis

We'll use the ATLAS 2010 W+jets analysis as an example. Feel free to implement something else: we'll try to troubleshoot.

The SPIRES key for this ATLAS analysis is 8919674 (try "key 8919674" in the SPIRES search box) and it was published in 2010, so in the standard Rivet naming convention it is called **`ATLAS_2010_S8919674`**.

There is reference data for this analysis in HepData: running **`rivet --show-analysis ATLAS_2010_S8919674`** supplies this URL: **http://hepdata.cedar.ac.uk/view/irn8919674**

**`rivet-mkanalysis ATLAS_2010_S8919674`** will download this ref data. NB. the jet multiplicity plots are not output correctly: HepData needs some improvements! Check the **`.info`** and **`.aida`** files: use **`aida2flat ATLAS_2010_S8919674.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)`**.

# UnstableFinalState

The `UnstableFinalState` projection fetches decayed-but-physical particles (mostly hadrons) from the event record. The HepMC standard declares how these are to be indicated, so the results are reliable and physically safe:

```
const UnstableFinalState ufs(2.5, 6.0);
addProjection(ufs, "UFS");
...
const FinalState& ufs =
  applyProjection<FinalState>(evt, "UFS");
foreach (const Particle& p, j.particles()) {
  const int pid = p.pdgId();
  if (PID::hasBottom(pid)) num_b += 1;
  ...
}
```

HepPDT-type functions are defined in the `PID` namespace in the `Rivet/Tools/ParticleIdUtils.hh`.

# The End