Practical Session 2

MC studies of $t\bar{t}$ reconstruction

IMPU-YITP Summer School 2011, Kyoto, 2011-09-07

Writing a Rivet analysis

Writing an analysis is of course more involved than just running **`rivet`**!

But the C++ programming interface is intended to be friendly: most analyses are quite short and simple because the bulk of the computation is in the library.

Key Rivet analysis features:

- 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 most Rivet computation 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.

Register projections with a name in the `init` method:

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

Projections – applying

Use the registered name to apply a projection to the current event:

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

Get a const reference to the applied projection 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 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.

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 `foreach`. We like the “make simple things simple” philosophy.

An aside: physics vectors

Rivet uses its own physics vectors rather than e.g. CLHEP. For the full interface see the Rivet 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! e.g. `myparticle.momentum().pT()`

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);
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` – useful for e.g. the lepton isolation.

Histogramming

Histograms are booked via helper methods on the **`Analysis`** base class, e.g. **`bookHistogram1D("thisname", 50, 0, 100)`**. Binnings can also be specified via a vector of bin edges (or *autobooked* from a reference histogram – not relevant today)

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

Plot presentation is specified in the **`.plot`** file accompanying the analysis. Directives include **`LogY=1`** (or **`=0`**), **`Title=foo`**, **`XLabel=bar`**, **`FullRange=1`**, ...

Today's analysis practical

You will be extending and optimising a Rivet analysis for semileptonic $t\bar{t}$ reconstruction: MC_TOP

The analysis method is to look for a hard lepton and missing E_T as a signature of the leptonically-decaying top. The remaining light and b-tagged jets are then used to reconstruct the other top.

You will have a signal and a background/inclusive event sample per generator. The analysis can be improved in many ways, e.g.

- More intelligent hadronic W reconstruction, e.g. tighter mass window, mass-constrained jet selection, lepton isolation…
- Use of extra variables for cuts, e.g. H_T, centrality